

# Last-use Opacity: A Strong Safety Property for Transactional Memory with Early Release Support

Konrad Siek, Paweł T. Wojciechowski  
Institute of Computing Science  
Poznań University of Technology

March 28, 2016

## Abstract

*Transaction Memory* (TM) is a concurrency control abstraction that allows the programmer to specify blocks of code to be executed atomically as transactions. However, since transactional code can contain just about any operation attention must be paid to the state of shared variables at any given time. E.g., contrary to a database transaction, if a TM transaction reads a stale value it may execute dangerous operations, like attempt to divide by zero, access an illegal memory address, or enter an infinite loop. Thus serializability is insufficient, and stronger safety properties are required in TM, which regulate what values can be read, even by transactions that abort. Hence, a number of TM safety properties were developed, including opacity, and TMS1 and TMS2. However, such strong properties preclude using early release as a technique for optimizing TM, because they virtually forbid reading from live transactions. On the other hand, properties that do allow early release are either not strong enough to prevent any of the problems mentioned above (recoverability), or add additional conditions on transactions with early release that limit their applicability (elastic opacity, live opacity, virtual world consistency). This paper introduces last-use opacity, a new TM safety property that is meant to be a compromise between strong properties like opacity and serializability. The property eliminates all but a small class of inconsistent views and poses no stringent conditions on transactions. For illustration, we present a last-use opaque TM algorithm and show that it satisfies the new safety property.

## 1 Introduction

Writing concurrent programs using the low-level synchronization primitives is notoriously difficult and error-prone. Over the past decade, there has been a growing interest in alternatives to lock-based synchronization by turning to the idea of software *transactional memory* (TM) [20, 28]. Basically, TM transplants the transaction abstraction from database systems and uses it to hide the details of synchronization. In particular, TM uses speculative execution to ensure that transactions in danger of reading inconsistent state abort and retry. This is a fairly universal solution and means that the programmer must only specify where transactions begin and end, and TM manages the execution so that the transactional code executes correctly and efficiently. Thus, the programmer avoids having to solve the problem of synchronization herself, and can rely on any one of a plethora of TM systems (e.g., [10, 17, 19, 18, 24, 27]).

Since TM allows transactional code to be mixed with non-transactional code and to contain virtually any operation, rather than just reads and writes like in its database predecessors, greater attention must be paid to the state of shared variables at any given time. For instance, if a database transaction reads a stale value, it must simply abort and retry, and no harm is done. Whereas, if a TM transaction reads a stale value it may execute an unanticipated dangerous operation, like dividing by zero, accessing an illegal memory address, or entering an infinite loop. Thus, TM systems must restrict the ability of transactions to view inconsistent state.

To that end, the safety property called opacity [14, 15] was introduced, which includes the condition that transactions do not read values written by other live (not completed) transactions alongside serializability [25] and real-time order conditions. Opacity became the gold standard of TM safety properties, and most TM systems found in the literature are, in fact, opaque. However opacity precludes *early release*, an important programming technique, where two transactions technically conflict but nevertheless both commit correctly, and still produce a history that is intuitively correct. Systems employing early release (e.g. [19, 26, 13, 8, 31]) show that this yields a significant and worthwhile performance benefit. This is particularly (but not exclusively) true with pessimistic concurrency control, where early release is vital to increased parallelism between transactions, and therefore essential for achieving high efficiency in applications with high contention.

Since opacity is a very restrictive property, a number of more relaxed properties were introduced that tweaked opacity’s various aspects to achieve a more practical property. These properties include virtual world consistency (VWC) [21], transactional memory specification (TMS1 and TMS2) [11], elastic opacity [13], and live opacity [12]. The first contribution of this paper is to examine these properties and determine whether or not they allow the use of early release in TM, and, if so, what compromises they make with respect to consistency, and what additional assumptions they require. We then consider the applicability of these properties to TM systems that rely on early release. In addition to TM properties, we similarly examine common database consistency conditions: serializability [25], recoverability [16], avoiding cascading aborts (ACA) [7], strictness [7], and rigorousness [9].

The second contribution of this paper is to introduce a new TM safety property called last-use opacity that allows early release without requiring stringent assumptions but nevertheless eliminates inconsistent views or restricts them to a manageable minimum. We give a formal definition, discuss example last-use opaque histories, and compare the new property with existing TM properties, specifically showing that it is stronger than serializability but weaker than opacity. We also describe the guarantees given by last-use opacity and consider the applicability of the property in system models that either allow, deny, or restrict the explicit programmatic abort operation. Whereas, last-use opacity eliminates inconsistent views in system models that forbid explicitly aborting transactions or restricts this to particular scenarios, we show that allowing free use of explicit aborts can lead to inconsistent views in last-use opaque histories. Thus, we also introduce a stronger variant of the property called  $\beta$ -last-use opacity that precludes them.

Finally, we give SVA [34], a TM concurrency control algorithm with early release and demonstrate that it satisfies last-use opacity.

The paper is structured as follows. We present the definitions of basic terms in Section 2. We follow by an examination of the TM property space in Section 3. Next, we define and discuss last-use opacity in Section 4. Then, we present SVA and demonstrate its correctness in Section 5. Finally, we present the related work in Section 6 and conclude in Section 7. We also include an appendix containing additional proofs.

## 2 Preliminaries

Before discussing properties and their relation to early release, let us provide definitions of the relevant ancillary concepts.

Let  $\Pi = \{p_1, p_2, \dots, p_n\}$  be a set of processes. Then, let program  $\mathbb{P}$  be defined as a set of subprograms  $\mathbb{P} = \{\mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_n\}$  such that for each process  $p_k$  in  $\Pi$  there is exactly one corresponding subprogram  $\mathcal{P}_k$  in  $\mathbb{P}$  and *vice versa*. Each subprogram  $\mathcal{P}_k \in \mathbb{P}$  is a finite sequence of statements in some language  $\mathbb{L}$ . The definition of  $\mathbb{L}$  can be whatsoever, as long as it provides constructs to execute operations on shared variables in accordance with the interface and assumptions described further in this section. In particular,  $\mathbb{L}$  can allow local computations whose effects are not visible outside a single processes.

Given program  $\mathbb{P}$  and a set of processes  $\Pi$ , we denote an execution of  $\mathbb{P}$  by  $\Pi$  as  $\mathcal{E}(\mathbb{P}, \Pi)$ . An execution entails each process  $p_k \in \Pi$  evaluating some prefix of subprogram  $\mathcal{P}_k \in \mathbb{P}$ . The evaluation of each statement by a process is deterministic and follows the semantics of  $\mathbb{L}$ .  $\mathcal{E}(\mathbb{P}, \Pi)$  is concurrent, i.e. while the statements in subprogram  $\mathcal{P}_k$  are evaluated sequentially by a single process, the evaluation of statements by different processes can be arbitrarily interleaved. We call  $\mathcal{E}(\mathbb{P}, \Pi)$  a *complete* execution if each process  $p_k$  in  $\Pi$  evaluates all of the statements in  $\mathcal{P}_k$ . Otherwise, we call  $\mathcal{E}(\mathbb{P}, \Pi)$  a *partial* execution.

**Variables** Let  $\mathbb{V}$  be a set of *shared variables* (or *variables*, in short). Each variable, denoted as  $x, y, z$  etc., supports the following *operations*, denoted  $o$ , that allow to retrieve or modify its state:

- a) *write* operation  $w(x)v$  that sets the state of  $x$  to value  $v$ ; the operation's *return value* is the constant *ok*,
- b) *read* operation  $r(x)$  whose *return value* is the current state of  $x$ .

In order to execute some operation  $o$  on variable  $x$ , process  $p_k$  issues an *invocation event* denoted  $inv^k(o)$ , and receives a *response event* denoted  $res^k(u)$ , where  $u$  is the return value of  $o$ . The pair of these events is called a *complete operation execution* and it is denoted  $o^k \rightarrow u$ , whereas an invocation event  $inv^k(o)$  without the corresponding response event is called a *pending operation execution*. Specifically, a complete execution of a read operation by process  $p_k$  is denoted  $r^k(x) \rightarrow v$  and a complete execution of a write operation is denoted  $w^k(x)v \rightarrow ok$ . We refer to complete and pending operation executions as *operation executions*, denoted by *op*.

Each event is atomic and instantaneous, but the execution of the entire operation composed of two events is not.

**Transactions** *Transactional memory (TM)* is a programming paradigm that uses transactions to control concurrent execution of operations on shared variables by parallel processes. A *transaction*  $T_i \in \mathbb{T}$  is some piece of code executed by process  $p_k$ , as part of subprogram  $\mathcal{P}_k$ . Hence, we say that  $p_k$  executes  $T_i$ . Process  $p_k$  can execute local computations as well as operations on shared variables as part of the transaction. In particular, the processes can execute the following operations as part of transaction  $T_i$ :

- a)  $start_i$  which initializes transaction  $T_i$ , and whose return value is the constant  $ok_i$ ,
- b)  $w_i(x)v$  and  $r_i(x)$  which respectively write a value  $v$  to variable  $x$  and read  $x$  within transaction  $T_i$ , and return either the operation's return value or the constant  $A_i$ ,

- c)  $tryC_i$  which attempts to commit  $T_i$  and returns either the constant  $C_i$  or the constant  $A_i$ .

There is also another operation allowed in some TM systems and not in others, and we wish to discuss it separately. Namely, some TMs allow for a transaction to programmatically roll back by executing the operation:

- d)  $tryA_i$  which aborts  $T_i$  and returns  $A_i$ .

The constant  $A_i$  indicates that transaction  $T_i$  has been aborted, as opposed to the constant  $C_i$  which signifies a successful commitment of the transaction.

By analogy to processes executing operations on variables, if process  $p_k$  executes some operation as part of transaction  $T_i$  it issues an invocation event of the form  $inv_i^k(start_i)$ ,  $inv_i^k(o)$  for some  $x$ , or  $inv_i^k(tryC_i)$ , (or possibly  $inv_i^k(tryA_i)$ ) and receives a response of the form  $res_i^k(u_i)$ , where  $u_i$  is a value, or the constant  $ok_i$ ,  $C_i$ , or  $A_i$ . The superscript always denotes which process executes the operation, and the subscript denotes of which transaction the operation is a part. We denote operation executions by process  $p_k$  within transaction  $T_i$  as:

- a)  $start_i^k \rightarrow ok_i$ ,
- b)  $r_i^k(x) \rightarrow v$  or  $r_i^k(x) \rightarrow A_i$ ,
- c)  $w_i^k(x)v \rightarrow ok_i$  or  $w_i^k(x)v \rightarrow A_i$ ,
- d)  $tryC_i^k \rightarrow C_i$  or  $tryC_i^k \rightarrow A_i$ .
- e)  $tryA_i^k \rightarrow A_i$ .

TM assumes that processes execute operations on shared variables only as part of a transaction. Furthermore, we assume that any transaction  $T_i$  is executed by exactly one process  $p_k$  and that each process executes transactions sequentially.

Even though transactions are subprograms evaluated by processes, it is convenient to talk about them as separate and independent entities. Thus, rather than saying  $p_k$  executes some operation as part of transaction  $T_i$ , we will simply say that  $T_i$  executes (or performs) some operation. Hence we will also forgo the distinction of processes in transactional operation executions, and write simply:  $start_i \rightarrow ok_i$ ,  $r_i(x) \rightarrow v$ ,  $w_i(x)v \rightarrow ok_i$ ,  $tryC_i \rightarrow C_i$ , etc. By analogy, we also drop the superscript indicating processes in the notation of invocation and response events, unless the distinction is needed.

**Sequential Specification** Given variable  $x$ , let *sequential specification* of  $x$ , denoted  $Seq(x)$ , be a prefix-closed set of sequences containing invocation events and response events which specify the semantics of shared variables. (A set  $Q$  of sequences is *prefix-closed* if, whenever a sequence  $S$  is in  $Q$ , every prefix of  $S$  is also in  $Q$ .) Intuitively, a sequential specification enumerates all possible correct sequences of operations that can be performed on a variable in a sequential execution. Specifically, given  $D$ , the domain of variable  $x$ , and  $v_0 \in D$ , an initial state of  $x$ , we denote by  $Seq(x)$  the sequential specification of  $x$  s.t.,  $Seq(x)$  is a set of sequences of the form  $[\alpha_1 \rightarrow v_1, \alpha_2 \rightarrow v_2, \dots, \alpha_m \rightarrow v_m]$ , where each  $\alpha_j \rightarrow v_j$  ( $j = 1..m$ ) is either:

- a)  $w_i(x)v_j \rightarrow ok_i$ , where  $v_j \in D$ , or

- b)  $r_i(x) \rightarrow v_j$ , and either the most recent preceding write operation is  $w_l(x)v_j \rightarrow ok_l$  ( $l < i$ ) or there are no preceding writes and  $v_j = v_0$ .

From this point on we assume that the domain  $D$  of all transactional variables is the set of natural numbers  $\mathbb{N}_0$  and that the initial value  $v_0$  of each variable is 0.

Even though we describe the interface and sequential specification of variables to represent the behavior of registers, we do so out of convenience and our conclusions can be trivially extended to other types of objects e.g. compare and swap objects or stacks.

**Histories** A TM *history*  $H$  is a sequence of invocation and response events issued by the execution of transactions  $\mathbb{T}_H = \{T_1, T_2, \dots, T_t\}$ . The occurrence and order of events in  $H$  is dictated by a given (possibly partial) execution of some program  $\mathbb{P}$  by processes  $\Pi$ . We denote by  $H \models \mathcal{E}(\mathbb{P}, \Pi)$  that history  $H$  is produced by  $\mathcal{E}(\mathbb{P}, \Pi)$ . Note, that different interleavings of processes in  $\mathcal{E}(\mathbb{P}, \Pi)$  can produce different histories. A *subhistory* of a history  $H$  is a subsequence of  $H$ .

The sequence of events in a history  $H_j$  can be denoted as  $H_j = [e_1, e_2, \dots, e_m]$ . For instance, some history  $H_1$  below is a history of a run of some program that executes transactions  $T_1$  and  $T_2$ :

$$H_1 = [ \text{inv}_1(\text{start}_1), \text{res}_1(ok_1), \text{inv}_2(\text{start}_2), \text{res}_2(ok_2), \\ \text{inv}_1(w_1(x)v), \text{inv}_2(r_2(x)), \text{res}_1(ok_1), \text{res}_2(v), \\ \text{inv}_1(\text{try}C_1), \text{res}_1(C_1), \text{inv}_2(\text{try}C_2), \text{res}_2(C_2) ].$$

Given any history  $H$ , let  $H|T_i$  be the longest subhistory of  $H$  consisting only of invocations and responses executed by transaction  $T_i$ . For example,  $H_1|T_2$  is defined as:

$$H_1|T_2 = [ \text{inv}_2(\text{start}_2), \text{res}_2(ok_2), \text{inv}_2(r_2(x)), \text{res}_2(v), \text{inv}_2(\text{try}C_2), \text{res}_2(C_2) ].$$

We say transaction  $T_i$  is in  $H$ , which we denote  $T_i \in H$ , if  $H|T_i \neq \emptyset$ .

Let  $H|p_k$  be the longest subhistory of  $H$  consisting only of invocations and responses executed by process  $p_k$ .

Let  $H|x$  be the longest subhistory of  $H$  consisting only of invocations and responses executed on variable  $x$ , but only those that form complete operation executions.

Given complete operation execution  $op$  that consists of an invocation event  $e'$  and a response event  $e''$ , we say  $op$  is in  $H$  ( $op \in H$ ) if  $e' \in H$  and  $e'' \in H$ . Given a pending operation execution  $op$  consisting of an invocation  $e'$ , we say  $op$  is in  $H$  ( $op \in H$ ) if  $e' \in H$  and there is no other operation execution  $op'$  consisting of an invocation event  $e'$  and a response event  $e''$  s.t.  $op' \in H$ .

Given two complete operation executions  $op'$  and  $op''$  in some history  $H$ , where  $op'$  contains the response event  $res'$  and  $op''$  contains the invocation event  $inv''$ , we say  $op'$  precedes  $op''$  in  $H$  if  $res'$  precedes  $inv''$  in  $H$ .

A history whose all operation executions are complete is a *complete* history.

Most of the time it will be convenient to denote any two adjoining events in a history that represent the invocation and response of a complete execution of an operation as that operation execution, using the syntax  $e \rightarrow e'$ . Then, an alternative representation of  $H_1|T_2$  is denoted as follows:

$$H_1|T_2 = [ \text{start}_2 \rightarrow ok_2, r_2(x) \rightarrow v, \text{try}C_2 \rightarrow C_2 ].$$

History  $H$  is *well-formed* if, for every transaction  $T_i$  in  $H$ ,  $H|T_i$  is an alternating sequence of invocations and responses s.t.,

- a)  $H|T_i$  starts with an invocation  $inv_i(start_i)$ ,
- b) no events in  $H|T_i$  follow  $res_i(C_i)$  or  $res_i(A_i)$ ,
- c) no invocation event in  $H|T_i$  follows  $inv_i(tryC_i)$  or  $inv_i(tryA_i)$ ,
- d) for any two transactions  $T_i$  and  $T_j$  s.t.,  $T_i$  and  $T_j$  are executed by the same process  $p_k$ , the last event of  $H|T_i$  precedes the first event of  $H|T_j$  in  $H$  or *vice versa*.

In the remainder of the paper we assume that all histories are well-formed.

**History Completion** Given history  $H$  and transaction  $T_i$ ,  $T_i$  is *committed* if  $H|T_i$  contains operation execution  $tryC_i \rightarrow C_i$ . Transaction  $T_i$  is *aborted* if  $H|T_i$  contains response  $res_i(A_i)$  to any invocation. Transaction  $T_i$  is *commit-pending* if  $H|T_i$  contains invocation  $tryC_i$  but it does not contain  $res_i(A_i)$  nor  $res_i(C_i)$ . Finally,  $T_i$  is *live* if it is neither committed, aborted, nor commit-pending.

Given two histories  $H' = [e'_1, e'_2, \dots, e'_m]$  and  $H'' = [e''_1, e''_2, \dots, e''_m]$ , we define their concatenation as  $H' \cdot H'' = [e'_1, e'_2, \dots, e'_m, e''_1, e''_2, \dots, e''_m]$ . We say  $P$  is a prefix of  $H$  if  $H = P \cdot H'$ . Then, let a *completion*  $Compl(H)$  of history  $H$  be any complete history s.t.,  $H$  is a prefix of  $Compl(H)$  and for every transaction  $T_i \in H$  subhistory  $Compl(H)|T_i$  equals one of the following:

- a)  $H|T_i$ , if  $T_i$  finished committing or aborting,
- b)  $H|T_i \cdot [res_i(C_i)]$ , if  $T_i$  is live and contains a pending  $tryC_i$ ,
- c)  $H|T_i \cdot [res_i(A_i)]$ , if  $T_i$  is live and contains some pending operation,
- d)  $H|T_i \cdot [tryC_i \rightarrow A_i]$ , if  $T_i$  is live and contains no pending operations.

Note that, if all transactions in  $H$  are committed or aborted then  $Compl(H)$  and  $H$  are identical.

Two histories  $H'$  and  $H''$  are *equivalent* (denoted  $H' \equiv H''$ ) if for every  $T_i \in \mathbb{T}$  it is true that  $H'|T_i = H''|T_i$ . When we write  $H'$  is equivalent to  $H''$  we mean that  $H'$  and  $H''$  are equivalent.

**Sequential and Legal Histories** A *real-time order*  $<_H$  is an order over history  $H$  s.t., given two transactions  $T_i, T_j \in H$ , if the last event in  $H|T_i$  precedes in  $H$  the first event of  $H|T_j$ , then  $T_i$  *precedes*  $T_j$  in  $H$ , denoted  $T_i <_H T_j$ . We then say that two transactions  $T_i, T_j \in H$  are *concurrent* if neither  $T_i <_H T_j$  nor  $T_j <_H T_i$ . We say that history  $H'$  *preserves the real-time order* of  $H$  if  $<_H \subseteq <_{H'}$ . A *sequential history*  $S$  is a history, s.t. no two transactions in  $S$  are concurrent in  $S$ . Some sequential history  $S$  is a *sequential extension* of  $H$  if  $S$  is equivalent to  $H$  and  $S$  preserves the real time order of  $H$ .

We analogously define a real-time order  $<_H$  of operation executions over history  $H$ .

Let  $S'$  be a sequential history that only contains committed transactions, with the possible exception of the last transaction, which can be aborted. We say that sequential history  $S'$  is *legal* if for every  $x \in \mathbb{V}$ ,  $S'|x \in Seq(x)$ .

Using the definitions above allows us to formulate the central concept that defines consistency in opacity: *transaction legality*. Intuitively, we can say a transaction is legal in a sequential history if it only reads values of variables that were written by committed transactions or by itself. More formally, given a sequential history  $S$  and a transaction  $T_i \in S$ ,

we then say that transaction  $T_i$  is *legal in  $S$*  if  $Vis(S, T_i)$  is legal, where  $Vis(S, T_i)$  is the longest subhistory  $S'$  of  $S$  s.t., for every transaction  $T_j \in S'$ , either  $i = j$  or  $T_j$  is committed in  $S'$  and  $T_j <_S T_i$ .

**Unique Writes** History  $H$  has *unique writes* if, given transactions  $T_i$  and  $T_j$  (where  $i \neq j$  or  $i = j$ ), for any two write operation executions  $w_i(x)v' \rightarrow ok_i$  and  $w_j(x)v'' \rightarrow ok_j$  it is true that  $v' \neq v''$  and neither  $v' = v_0$  nor  $v'' = v_0$ .

For the remainder of the paper we focus exclusively on histories with unique writes. This assumption does not reduce generality, in that any history without unique writes trivially can be transformed into a history with unique writes (for instance, by appending a timestamp to each written value).

**Accesses** Given a history  $H$  and a transaction  $T_i$  in  $H$ , we say that  $T_i$  *reads* variable  $x$  in  $H$  if there exists an invocation  $inv_i(r_i(x))$  in  $H|T_i$ . By analogy, we say that  $T_i$  *writes* to  $x$  in  $H$  if there exists an invocation  $inv_i(w_i(x)v)$  in  $H|T_i$ . If  $T_i$  reads  $x$  or writes to  $x$  in  $H$ , we say  $T_i$  *accesses*  $x$  in  $H$ . In addition, let  $T_i$ 's *read set* be a set that contains every variable  $x$ , s.t.  $T_i$  reads  $x$ . By analogy,  $T_i$ 's *write set* contains every  $x$ , s.t.  $T_i$  writes to  $x$ . A transaction's *access set*, denoted  $ASet(T_i)$ , is the union of its read set and its write set.

Given a history  $H$  and a pair of transactions  $T_i, T_j \in H$ , we say  $T_i$  and  $T_j$  *conflict* on variable  $x$  in  $H$  if  $T_i$  and  $T_j$  are concurrent, both  $T_i$  and  $T_j$  access  $x$ , and one or both of  $T_i$  and  $T_j$  write to  $x$ .

Given a history  $H$  (with unique writes) and a pair of transactions  $T_i, T_j \in H$ , we say  $T_i$  *reads from*  $T_j$  if there is some variable  $x$ , for which there is a complete operation execution  $w_j(x)v \rightarrow ok_j$  in  $H|T_j$  and another complete operation execution  $r_i(x) \rightarrow u$  in  $H|T_i$ , s.t.  $v = u$ .

Given any transaction  $T_i$  in some history  $H$  (with unique writes) any operation execution on a variable  $x$  within  $H|T_i$  is either *local* or *non-local*. Read operation execution  $r_i(x) \rightarrow v$  in  $H|T_i$  is local if it is preceded in  $H|T_i$  by a write operation execution on  $x$ , and it is non-local otherwise. Write operation execution  $w_i(x)v \rightarrow ok_i$  in  $H|T_i$  is local if it is followed in  $H|T_i$  by an invocation of a write operation on  $x$ , and non-local otherwise.

**Safety Properties** A *property*  $\mathfrak{P}$  is a condition that stipulates correct behavior. In relation to histories, a given history satisfies  $\mathfrak{P}$  if the condition is met for that history. In relation to programs, program  $\mathbb{P}$  satisfies  $\mathfrak{P}$  if all histories produced by  $\mathbb{P}$  satisfy  $\mathfrak{P}$ .

Safety properties [23] are properties which guarantee that "something [bad] will not happen." In the case of TM this means that, transactions will not observe concurrency of other transactions. Property  $\mathfrak{P}$  is a safety property if it meets the following definition (adapted from [5]):

**Definition 1.** A *property*  $\mathfrak{P}$  is a safety property if, given the set  $\mathbb{H}_{\mathfrak{P}}$  of all histories that satisfy  $\mathfrak{P}$ :

- a) Prefix-closure: every prefix  $H'$  of a history  $H \in \mathbb{H}_{\mathfrak{P}}$  is also in  $\mathbb{H}_{\mathfrak{P}}$ ,
- b) Limit-closure: for any infinite sequence of finite histories  $H_0, H_1, \dots$ , s.t. for every  $H_h \in \mathbb{H}_{\mathfrak{P}}$  and  $H_h$  is a prefix of  $H_{h+1}$ , the infinite history that is the limit of the sequence is also in  $\mathbb{H}_{\mathfrak{P}}$ .

For distinction, in the remainder of the paper we refer to properties that are not safety properties as *consistency conditions*.

### 3 Early Release

In this section we discuss whether existing safety properties and consistency conditions allow for early release (extending our work in [33]) and to what extent. The aim of the analysis is to find properties that describe the guarantees of TM systems with early release that can be applied in practice. That is, we seek a safety property that allows early release but reduces or eliminates undesired behaviors.

Early release pertains to a situation where conflicting transactions execute partially in parallel while accessing the same variable. The implied intent is for all such transactions to access these variables without losing consistency and thus for them all to finally commit. We define the concept of early release as follows:

**Definition 2** (Early Release). *Given history  $H$  (with unique writes), transaction  $T_i \in H$  releases variable  $x$  early in  $H$  iff there is some prefix  $P$  of  $H$ , such that  $T_i$  is live in  $P$  and there exists some transaction  $T_j \in P$  such that there is a complete non-local read operation execution  $op_j = r_j(x) \rightarrow v$  in  $P|T_j$  and a write operation execution  $op_i = w_i(x)v \rightarrow ok_i$  in  $P|T_i$  such that  $op_i$  precedes  $op_j$  in  $P$ .*

We begin our analysis by defining its key questions. The first and the most obvious is whether a particular property supports early release at all. This is defined as follows:

**Definition 3** (Early Release Support). *Property  $\mathfrak{P}$  supports early release iff given some history  $H$  that satisfies  $\mathfrak{P}$  there exists some transaction  $T_i \in H$ , s.t.  $T_i$  releases some variable  $x$  early in  $H$ .*

If a property allows early release, it allows a significant performance boost (e.g. [26, 31]) as transactions are executed with a higher degree of parallelism. However, early release can give rise to some unwanted or unintuitive scenarios with respect to consistency. The most egregious of these is *overwriting*, where one transaction releases some variable early, but proceeds to modify it afterward. In that case, any transaction that started executing operations on the released variable will observe an intermediate value with respect to the execution of the other transaction, ie., *view inconsistent state*.

An example of overwriting is shown in Fig. 1, where transaction  $T_i$  releases variable  $x$  early but continues to write to  $x$  afterward. As a consequence,  $T_j$  first reads the value of  $x$  that is later modified. When  $T_j$  detects it is in conflict while executing a write operation it is aborted. This is a way for the TM to attempt to mitigate the consequences of viewing inconsistent state. The transaction is then restarted as a new transaction  $T_{j'}$ .

However, as argued in [15], simply aborting a transaction that views inconsistent state is not enough, since the transaction can potentially act in an unpredictable way on the basis of using an inconsistent value to perform local operations. For instance, if the value is used in pointer arithmetic it is possible for the transaction to access an unexpected memory location and crash the process. Alternatively, if the transaction uses the value within a loop condition, it can enter an infinite loop and become parasitic.

Thus, in our analysis of existing properties we ask the question whether, apart from allowing early release, the properties also forbid overwriting. In the light of the potential dangerous behaviors that can be caused by it, we consider properties that allow overwriting to be too weak to be practical.

**Definition 4** (Overwriting Support). *Property  $\mathfrak{P}$  supports overwriting iff  $\mathfrak{P}$  supports early release, and given some history  $H$  (with early release) that satisfies  $\mathfrak{P}$ , for some pair of transactions  $T_i, T_j \in H$  s.t.,*

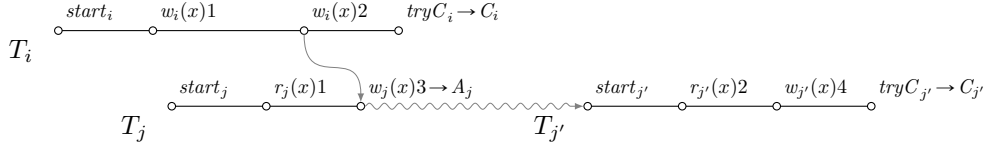


Figure 1: History with early release and overwriting. The diagram depicts some history  $H$  presented as operations executed by transactions on a time axis. Every line depicts the operations executed by a particular transaction, e.g., the line marked  $T_i$  depicts subhistory  $H|T_i$ . The symbol  $\text{—}\circ\text{—}$  denotes a complete operation execution (an invocation event immediately followed by a response event). For brevity, whenever the response event of some operation execution is  $ok_i$  we omit it, eg., we write  $w_i(x)1$  rather than  $w_i(x)1 \rightarrow ok_i$ . We also shorten the representation of complete read operation executions, so that eg.  $r_j(x) \rightarrow 1$  is represented as  $r_j(x)1$ . The arrow  $\curvearrowright$  is used to emphasize a happens before relation, and  $\sim\sim\sim$  denotes that the preceding transaction aborts (here,  $T_j$ ) and a new transaction ( $T_{j'}$ ) is spawned.

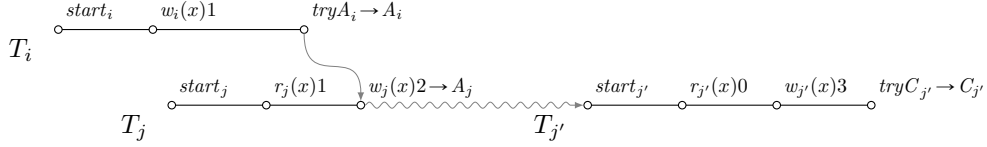


Figure 2: History with early release and cascading abort.

- a)  $T_i$  releases some variable  $x$  early,
- b)  $H|T_i$  contains two write operation executions:  $w_i(x)v \rightarrow ok_i$  and  $w_i(x)v' \rightarrow ok_i$ , s.t. the former precedes the latter in  $H|T_i$ ,
- c)  $H|T_j$  contains a read operation execution  $r_j(x) \rightarrow v$  that precedes  $w_i(x)v' \rightarrow ok_i$  in  $H$ .

In addition, we look at whether or not a particular property forbids a transaction that releases some variable early to abort. This is a precaution taken by many properties to prevent *cascading aborts*, another type of scenario leading to inconsistent views. An example of this is shown in Fig. 2. In such a case a transaction, here  $T_i$ , releases a variable early and subsequently aborts. This can cause another transaction  $T_j$  that executed operations on that variable in the meantime to observe inconsistent state. In order to maintain consistency, a TM will then typically force  $T_j$  to abort and restart as a result.

However, while the condition that no transaction that releases early can abort, solves the problem of cascading aborts, it significantly limits the usefulness of any TM that satisfies it, since TM systems typically cannot predict whether any particular transaction eventually commits or aborts. In particular, there are important applications for TM, where a transaction can arbitrarily and uncontrollably abort at any time. Such applications include distributed TM and hardware TM, where aborts can be caused by outside stimuli, such as machine crashes.

An exception to this may be found in systems making special provisions to ensure that irrevocable transactions eventually commit (see e.g., [37]). In such systems, early release

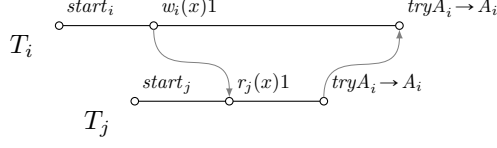


Figure 3: History with an aborting early release transaction.

transactions could be ensured never to abort. However, case in point, these take drastic measures to ensure that, e.g., at most a single irrevocable transaction is present in the system at one time. Therefore, the requirement may be difficult to enforce.

Finally, the requirement that transactions which released early must not abort precludes some scenarios that are intuitively correct. For instance, take the example in Fig. 3. Here,  $T_i$  writes 1 to  $x$  and releases it early.  $T_j$  reads 1 from  $x$  and then aborts by executing the  $tryA$  operation, which also causes  $T_i$  to abort. Since  $T_j$  reads from  $T_i$  while the latter is live,  $T_i$  releases early in this history. Then, if there is a requirement that transactions which release early not abort, then this history is an incorrect one. However, since  $T_j$  aborted on its own accord, there are no transactions that would be affected by  $T_i$  aborting later on. Hence, intuitively, the history is actually correct. Thus, we consider the requirement that transactions which release early must not abort to be overstrict.

Hence, we seek properties that allow aborts in transactions that release early.

**Definition 5** (Aborting Early Release Support). *Property  $\mathfrak{P}$  supports aborting early release iff  $\mathfrak{P}$  supports early release, and given some history  $H$  that satisfies  $\mathfrak{P}$ , for some transaction  $T_i \in H$  that releases some variable  $x$  early,  $H|T_i$  contains  $A_i$ .*

The properties under consideration are the typical TM safety properties: serializability, opacity, transactional memory specification, virtual world consistency, and elastic opacity. Furthermore, we examine some of the family of live properties from [12], since this recent work introduces a number of relaxed versions of TM safety properties with the view of accommodating early release. Finally, we consider some strong database consistency conditions that pertain to transactional processing: recoverability, avoiding cascading aborts, strictness, and rigorousness.

### 3.1 Serializability

The first property we consider is serializability, which can be regarded as a baseline TM safety property. It is defined in [25] in three variants: conflict serializability, view serializability, and final-state serializability. We follow a more general version of serializability defined in [36] (as *global atomicity*), which we adjust to account for non-atomicity of commits in our model.

**Definition 6** (Serializability). *History  $H$  is serializable iff there exists some sequential history  $S$  equivalent to a completion  $Compl(H)$  such that any committed transaction in  $S$  is legal in  $S$ .*

This definition does not preclude early release, as long as illegal transactions are aborted. Serializability also permits overwriting and cascading aborts.

**Theorem 1.** *Serializability supports early release.*

*Proof.* Let  $H$  be a transactional history as shown in Fig. 1. Note that since all transactions in  $H$  are committed or aborted then  $H = \text{Compl}(H)$ . Then, let there be a sequential history  $S = H|T_i \cdot H|T_j \cdot H|T_{j'}$ . Note that  $S \equiv H$ . Trivially, all the committed transactions in  $S$ , i.e.  $T_i$  and  $T_{j'}$ , are legal in  $S$ , so  $H$  is serializable. Since, by Def. 2,  $T_i$  releases early in  $H$ , then, by Def. 3, serializability supports early release.  $\square$

**Theorem 2.** *Serializability supports overwriting.*

*Proof.* Let  $H$  be a serializable history as in the proof of Theorem 1 above. Transaction  $T_i$  writes 1 to  $x$  in  $H$  prior to  $T_j$  reading 1 from  $x$ , and subsequently  $T_i$  writes 2 to  $x$ . Thus, according to Def. 4, serializability supports overwriting.  $\square$

**Theorem 3.** *Serializability supports aborting early release.*

*Proof.* Let  $H$  be a history such as the one in Fig. 2. Since all transactions in  $H$  are committed or aborted then  $H = \text{Compl}(H)$ . Then, let  $S$  be a sequential history equivalent to  $H$  such that  $S = H|T_i \cdot H|T_j \cdot H|T_{j'}$ .  $S$  contains only one committed transaction  $T_{j'}$ , which is trivially legal in  $S$ . Thus  $H$  is serializable. In addition, transaction  $T_i$  in  $S$  both releases  $x$  early (Def. 2) and contains an abort ( $A_i \in H|T_i$ ). Thus, by Def. 5, serializability supports aborting early release.  $\square$

### 3.2 Opacity

Opacity [14, 15] can be considered the standard TM safety property that guarantees serializability and preservation of real-time order, and prevents reading from live transactions. It is defined by the following two definitions. The first definition specifies *final state opacity* that ensures the appropriate guarantees for a complete transactional history. The second definition uses final state opacity to define a safety property that is prefix closed. Both definitions follow those in [15].

**Definition 7** (Final state opacity). *A finite TM history  $H$  is final-state opaque if, and only if, there exists a sequential history  $S$  equivalent to any completion of  $H$  s.t.,*

- (a)  $S$  preserves the real-time order of  $H$ ,
- (b) every transaction  $T_i$  in  $S$  is legal in  $S$ .

**Definition 8** (Opacity). *A TM history  $H$  is opaque if, and only if, every finite prefix of  $H$  is final-state opaque.*

**Theorem 4.** *Opacity does not support early release.*

*Proof.* By contradiction let us assume that opacity supports early release. Then, from Def. 3, there exists some history  $H$  (with unique writes), s.t.  $H$  is opaque and there exists some transaction  $T_i \in H$  that releases some variable  $x$  early in  $H$ .

From Def. 2, this implies that there exists some prefix  $P$  of  $H$  s.t.

- a) there is an operation execution  $op_i = w_i(x)v \rightarrow ok_i$  and  $op_i \in P|T_i$ ,
- b) there exists a transaction  $T_j \in P$  ( $i \neq j$ ) and an operation execution  $op_j = r_j(x) \rightarrow v$ , s.t.  $op_j \in P|T_j$  and  $op_i$  precedes  $op_j$  in  $P$ ,

c)  $T_i$  is live in  $P$ .

Let  $P_c$  be any completion of  $P$ . Since  $T_i$  is live in  $P$ , by definition of completion, it is necessarily aborted in  $P_c$  (ie.  $A_i \in P_c|T_i$ ). Given any sequential history  $S$  equivalent to  $P_c$ , since  $T_i$  is aborted in  $P_c$  and  $Vis(S, T_j)$  only contains operations of committed transactions, then  $P_c|T_i \not\subseteq Vis(S, T_j)$ . This means that  $op_j \in Vis(S, T_j)$  but  $op_i \notin Vis(S, T_j)$ , so  $Vis(S, T_j) \not\subseteq Seq(x)$  and therefore  $Vis(S, T_j)$  is not legal.

On the other hand, Def. 8 implies that any prefix  $P$  of  $H$  is final state opaque, which, by Def. 7, implies that there exists some completion  $P_c$  of  $P$  for which there exists an equivalent sequential history  $S$  s.t., any  $T_j$  in  $S$  is legal in  $S$ . Since any  $T_j$  is legal then for any  $T_j$ ,  $Vis(S, T_j)$  is legal. This is a contradiction with the paragraph above. Thus, there cannot exist a history like  $H$  that is both opaque and contains a transaction that releases some variable early.  $\square$

Since both Def. 4 and Def. 5 require early release support, then:

**Corollary 1.** *Opacity does not support overwriting.*

**Corollary 2.** *Opacity does not support aborting early release.*

### 3.3 TMS1 and TMS2

In [11] the authors argue that some scenarios, such as sharing variables between transactional and non-transactional code, require additional safety properties. Thus, they propose and rigorously define two consistency conditions for TM: *transactional memory specification 1 (TMS1)* and *transactional memory specification 2 (TMS2)*.

TMS1 follows a set of design principles including a requirement for observing consistent behavior that can be justified by some serialization. Among others, TMS1 also requires that partial effects of transactions are hidden from other transactions. These principles are reflected in the definition of the TMS1 automaton, and we paraphrase the relevant parts of the condition for the correctness of an operation's response in the following definitions (see the definitions of *extConsPrefix* and *validResp* for TMS1 in [11]).

Given a history  $H$  and some response event  $r$  in  $H$ , let  $H \uparrow r$  denote a subhistory of  $H$  s.t. for every operation execution  $op \in H$ ,  $op \in H \uparrow r$  iff  $op <_H r$  and  $op$  is complete. This represents all operations executed „thus far,” when considering the legality of  $r$ .

Let  $\mathbb{T}_H^d$  be the set of all transactions in  $H$  s.t.  $T_k \in \mathbb{T}_H^d$  iff  $T_k \in H$  and  $inv_k(tryC_k) \in H|T_k$ . Given response event  $r$ , let  $\mathbb{T}_H^d \uparrow r$  be the set of all transactions in  $H$  s.t.  $T_k \in \mathbb{T}_H^d \uparrow r$  if  $T_k \in \mathbb{T}_H^d$  and  $inv_k(tryC_k) <_H r$ . These sets represent transactions which committed or aborted (but are not live) and the set of all such transactions that did so before response event  $r$ .

Given some history  $H$ , let  $\mathbb{T}'_H$  be any subset of transactions in  $H$ . Let  $\sigma$  be a sequence of transactions. Let  $ser(\mathbb{T}'_H, <_H)$  be a set of all sequences of transactions s.t.  $\sigma \in ser(\mathbb{T}'_H, <_H)$  if  $\sigma$  contains every element of  $\mathbb{T}'_H$  exactly once and for any  $T_i, T_j \in \mathbb{T}'_H$ , if  $T_i <_H T_j$  then  $T_i$  precedes  $T_j$  in  $\sigma$ .

Given a history  $H$  and some response event  $r$  in  $H$ , let  $ops(\sigma, r)$  be a sequence of operations s.t. if  $\sigma = [T_1, T_2, \dots, T_n]$  then  $ops(\sigma, r) = H \uparrow r|T_1 \cdot H \uparrow r|T_2 \cdot \dots \cdot H \uparrow r|T_n$ . This represents the sequential history prior to response event  $r$  that respects a specific order of transactions defined by  $\sigma$ .

The most relevant condition in TMS1 with respect to early release checks the validity of individual response operations. A prerequisite for checking validity is to check whether

a response event can be justified by some *externally consistent prefix*. This prefix consists of operations from all transactions that precede the response event and whose effects are visible to other transactions. Specifically, if a transaction precedes another transaction in the real time order, then it must be both committed and included in the prefix, or both not committed and excluded from the prefix. However, if a transaction does not precede another transaction, it can be in the prefix regardless of whether it committed or aborted.

**Definition 9** (Extended Consistent Prefix). *Given a history  $H$  and a response event  $r$ , let the set of transactions  $\mathbb{T}_H^r$  be any subset of all transactions in  $H$  s.t. for any  $T_i, T_j \in \mathbb{T}_H^r$ , if  $T_i <_H T_j$  then  $T_i$  is in  $\mathbb{T}_H^r$  iff  $res_i(C_i) \in H \uparrow r | T_i$ .*

TMS1 specifies that each response to an operation invocation in a safe history must be *valid*. Intuitively, a valid response event is one for which there exists a sequential prefix that is both legal and meets the conditions of an externally consistent prefix. More precisely, the following condition must be met.

**Definition 10** (Valid Response). *Given a transaction  $T_i$  in  $H$ , we say the response  $r$  to some operation invocation  $e$  is valid if there exists set  $\mathbb{T}_H^r \subseteq \mathbb{T}_d \uparrow r$  and sequence  $\sigma \in ser(\mathbb{T}_H^r, <_H)$  s.t.  $\mathbb{T}_H^r$  satisfies Def. 9 and  $ops(\sigma \cdot T_i, r) \cdot [e \rightarrow r]$  is legal.*

**Theorem 5.** *TMS1 does not support early release.*

*Proof.* Assume by contradiction that TMS1 supports early release. Then by Def. 3, there exists some TMS1 history  $H$  s.t.  $T_i, T_j \in H$  and there is a prefix  $P$  of  $H$  s.t.  $op_i = w_i(x)v \rightarrow ok_i \in P | T_i$ ,  $op_j = r_j(x) \rightarrow v \in P | T_j$ , and  $T_i$  is live in  $H$ . This implies that  $inv_i(tryC_i) \notin P \uparrow res_j(v) | T_i$ . This means that  $T_i \notin \mathbb{T}_d$  and therefore not in any  $\mathbb{T}' \subseteq \mathbb{T}_d$  or, by extension, any  $\sigma \in ser(\mathbb{T}', <_H)$ . Therefore, there is no  $op_i$  in  $ops(\sigma, res_j(v))$ , so, assuming unique writes,  $op_j$  is not preceded by a write of  $v$  to  $x$  in  $ops(\sigma \cdot T_j, res_j(v)) \cdot [r_j(x) \rightarrow v]$ . Therefore,  $ops(\sigma \cdot T_j, res_j(v)) \cdot [r_j(x) \rightarrow v]$  is not legal, which contradicts Def. 10.  $\square$

Since both Def. 4 and Def. 5 require early release support, then:

**Corollary 3.** *TMS1 does not support overwriting.*

**Corollary 4.** *TMS1 does not support aborting early release.*

TMS2 is a stricter, but more intuitive version of TMS1. Since the authors show in [11] that TMS2 is strictly stronger than TMS1 (TMS2 implements TMS1), the conclusions above equally apply to TMS2. Hence, from Theorem 5:

**Corollary 5.** *TMS2 does not support early release.*

**Corollary 6.** *TMS2 does not support overwriting.*

**Corollary 7.** *TMS2 does not support aborting early release.*

### 3.4 Virtual World Consistency

The requirements of opacity, while very important in the context of TM's ability to execute any operation transactionally, can often be excessively stringent. On the other hand serializability is considered too weak for many TM applications. Thus, a weaker TM consistency condition called *virtual world consistency* (VWC) was introduced in [21]. The definition of

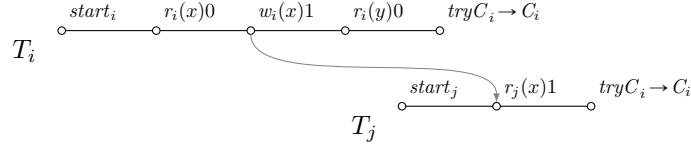


Figure 4: VWC history with early release.

VWC depends on *causal past*. The causal past  $C(H, T_i)$  of some transaction  $T_i$  in some history  $H$  is the set that contains  $T_i$  and all aborted or committed transactions that precede  $T_i$  in  $H$ . A causal past  $C(H, T_i)$  is legal, if for every  $T_j \in C(H, T_i)$ , s.t.  $i \neq j$ ,  $T_j$  is committed in  $H$ .

**Definition 11** (Virtual World Consistency). *History  $H$  is virtual world consistent iff all committed transactions are serializable and preserve real-time order, and for each aborted transaction there exists a linear extension of its causal past that is legal.*

This property allows a limited support for early release as follows.

**Theorem 6.** *VWC supports early release.*

*Proof.* Let  $H$  be a transactional history as shown in Fig. 4. Here,  $T_i$  performs two operations on  $x$  and one on  $y$ , while  $T_j$  reads  $x$ . The linear extension of  $H$  is  $S = H|T_i \cdot H|T_j$  wherein both transactions are trivially legal. Thus  $H$  is VWC. Since, by Def. 2,  $T_i$  releases early in  $H$ , then, by Def. 3, VWC supports early release.  $\square$

**Theorem 7.** *VWC does not support overwriting.*

*Proof.* Since VWC requires that aborting transactions view a legal causal past, then if a transaction reading  $x$  is aborted, it must read a legal (i.e. "final") value of  $x$ . Thus, let us consider some history  $H$  where some  $T_i$  releases  $x$  early, and some  $T_j$  reads  $x$  from  $T_i$ .

- a) If  $T_i$  writes to  $x$  after releasing it, and  $T_j$  commits, then  $T_j$  is not legal, and therefore  $H$  does not satisfy VWC.
- b) If  $T_i$  writes to  $x$  after releasing it, and  $T_j$  aborts, then the causal past of  $T_j$  contains  $T_i$ , and  $T_j$  reads an illegal (stale) value of  $x$  from  $T_i$ , so  $H$  does not satisfy VWC.

Therefore, any history  $H$  containing  $T_i$ , such that  $T_i$  releases  $x$  early and modifies it after release does not satisfy VWC. Hence, by Def. 4, VWC does not support overwriting.  $\square$

While VWC supports early release, there are severe limitations to this capability. That is, VWC does not allow a transaction that released early to subsequently abort for any reason.

**Theorem 8.** *VWC does not support aborting early release*

*Proof.* Given a history  $H$  that satisfies VWC and a transaction  $T_i \in H$  that releases variable  $x$  in  $H$ , let us assume for the sake of contradiction that  $T_i$  eventually aborts. By Def. 2, there is some  $T_j$  in  $H$  that reads from  $T_i$ . If  $T_i$  eventually aborts, then  $T_j$  reads from an aborted transaction.

- a) If  $T_j$  eventually aborts, then its causal past contains two aborted transactions ( $T_i$  and  $T_j$ ) and is, therefore, illegal. Hence  $H$  does not satisfy VWC, which is a contradiction.

b) If  $T_j$  eventually commits, then the sequential witness history is also illegal. Hence  $H$  does not satisfy VWC, which is a contradiction.

Therefore, if  $T_i$  eventually aborts,  $H$  does not satisfy VWC, which is a contradiction. Thus, since a VWC history cannot contain an abortable transaction that releases a variable early. Hence, by Def. 5, VWC does not support aborting early release.  $\square$

VWC does not allow for transactions that release early to abort, which we consider to be an impractical assumption in some TM systems and an overstrict requirement in general.

### 3.5 Live Opacity

*Live opacity* was introduced in [12] as part of a set of consistency conditions and safety properties that were meant to regulate the ability of transactions to read from live transactions. The work analyzes a number of properties and for each one presents a commit oriented variant that forbids early release and a live variant that allows it. Here, we concentrate on live opacity, since it best fits alongside the other properties presented here, however our conclusions will apply to the remainder of live properties.

Let  $H|(T_i, r)$  be the longest subsequence of  $H|T_i$  containing only read operation executions (possibly pending), with the exclusion of the last read operation if its response event is  $A_i$ . Let  $H|(T_i, gr)$  be a subsequence of  $H|(T_i, r)$  that contains only non-local operation executions. Let  $T_i^r$  be a transaction that invokes the same transactional operations as those invoked in  $H|(T_i, r) \cdot [inv_i(tryC_i)]$  if  $H|(T_i, r) \neq \emptyset$ , or  $\emptyset$  otherwise. Let  $T_i^{gr}$  be a transaction that invokes the same transactional operations as those invoked in  $[start_i \rightarrow ok_i] \cdot H|(T_i, gr) \cdot [tryC_i \rightarrow C_i]$  if  $H|(T_i, gr) \neq \emptyset$ , or  $\emptyset$  otherwise.

Given a history  $H$ , a transaction  $T_i \in H$ , and a complete local operation execution  $op = r_i(x) \rightarrow v$ , we say the latter's response event  $res_i(v)$  is *legal* if the last preceding write operation in  $H|T_i$  writes  $v$  to  $x$ . We say sequential history  $S$  justifies the serializability of history  $H$  when there exists a history  $H'$  that is a subsequence of  $H$  s.t.  $H'$  contains invocation and response events issued and received by transactions committed in  $H$ , and  $S$  is a legal history equivalent to  $H'$ .

**Definition 12** (Live Opacity). *A history  $H$  is live opaque iff, there exists a sequential history  $S$  that preserves the real time order of  $H$  and justifies that  $H$  is serializable and all of the following hold:*

- a) *We can extend history  $S$  to get a sequential history  $S'$  such that:*
  - *for each transaction  $T_i \in H$  s.t.  $T_i \notin S$ ,  $T_i^{gr} \in S'$ ,*
  - *if  $<$  is a partial order induced by the real time order of  $S$  in such a way that for each transaction  $T_i \in H$  s.t.  $T_i \notin S$  we replace each instance of  $T_i$  in the set of pairs of the real time order of  $H$  with transaction  $T_i^{gr}$ , then  $S'$  respects  $<$ ,*
  - *$S'$  is legal.*
- b) *For each transaction  $T_i \in H$  s.t.  $T_i \notin S$  and for each operation  $op$  in  $T_i^r$  that is not in  $T_i^{gr}$ , the response for  $op$  is legal.*

**Theorem 9.** *Live opacity supports early release.*

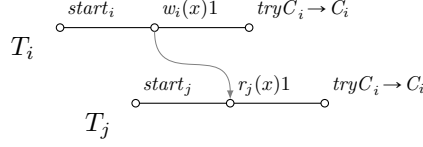


Figure 5: Live opaque history with early release.

*Proof.* Let history  $H$  be that represented in Fig. 5. Since there is a transaction  $T_i \in H$  that writes 1 to  $x$  and a transaction  $T_j$  that reads 1 from  $x$  before  $T_i$  commits, then there is a prefix  $P$  of  $H$  that meets Def. 2. Therefore  $T_i$  releases  $x$  early in  $H$ .

Let  $S$  be a sequential history s.t.  $S = H|T_i \cdot H|T_j$ . Since the real-time order of  $H$  is  $\emptyset$ , then, trivially,  $S$  preserves the real-time order of  $H$ . Since  $Vis(S, T_i)$  contains only  $H|T_i$  and therefore only a single write operation execution and no reads, then it is legal and  $T_i$  in  $S$  is legal in  $S$ . Furthermore,  $Vis(S, T_j)$  is such that  $Vis(S, T_j) = H|T_i \cdot H|T_j$  and contains a read operation  $r_j(x) \rightarrow 1$  preceded by the only write operation  $w_i(x)1 \rightarrow ok_i$ , so  $Vis(S, T_j)$  is legal, and, consequently,  $T_j$  in  $S$  is legal in  $S$ . Thus, all transactions in  $S$  are legal in  $S$ , so  $H$  is serializable.

Let  $S'$  be a sequential history that extends  $S$  in accordance to Def. 12. Since there are no transactions in  $S'$  that are not in  $S$ , then  $S' = S$ . Thus, since every transaction in  $S$  is legal in  $S$ , then every transaction in  $S'$  is legal in  $S'$ . Trivially,  $S'$  also preserves the real time order of  $S$ . Therefore, the condition Def. 12a is met. Since there are no local read operations in  $S$ , then condition Def. 12b is trivially met as well. Therefore,  $H$  is live opaque.

Since  $H$  is both live opaque and contains a transaction that releases early, then the theorem holds.  $\square$

**Theorem 10.** *Live opacity does not support overwriting.*

*Proof.* For the sake of contradiction, let us assume that there is a history (with unique writes)  $H$  that is live opaque and, from Def. 4, contains some transaction  $T_i$  that writes value  $v$  to some variable  $x$  and releases  $x$  early and subsequently executes another write operation writing  $v'$  to  $x$  where the second write follows a read operation executed by transaction  $T_j$  reading  $v$  from  $x$ .

Since  $H$  is live opaque there exists a sequential history  $S$  that justifies the serializability of  $H$ . There cannot exist a sequential history  $S$  where  $T_j$  reads from  $x$  between two writes to  $x$  executed by  $T_i$ , because there cannot exist a legal  $Vis(S, T_j)$ , so  $T_j$  would not be legal in  $S$ . Therefore,  $T_j$  must be aborted in  $H$  and therefore  $T_j$  is not in any sequential history  $S$  that justifies the serializability of  $H$ .

Since  $T_j$  is in  $H$  but not in  $S$ , then given any sequential extension  $S'$  of  $S$  in accordance to Def. 12  $T_j$  is replaced in  $S'$  by  $T_j^{gr}$  which reads  $v$  from  $x$  and finally commits. However, since the write operation execution writing  $v$  to  $x$  in  $T_i$  is followed in  $S'|T_i$  by another write operation execution that writes  $v'$  to  $x$ , then there cannot exist a  $Vis(S', T_j^{gr})$  that is legal. Thus  $T_j^{gr}$  in  $S'$  cannot be legal in  $S'$ , which contradicts Def. 12a. Thus,  $H$  is not live opaque, which is a contradiction.

Therefore  $H$  cannot simultaneously be live opaque and contain a transaction with early release and overwriting.  $\square$

**Theorem 11.** *Live opacity does not support aborting early release.*

*Proof.* For the sake of contradiction, let us assume that there is a history (with unique writes)  $H$  that is live opaque and, from Def. 5, contains some transaction  $T_i$  that writes value  $v$  to some variable  $x$  and releases  $x$  and subsequently aborts in  $H$ .

Let  $S$  be any sequential history that justifies the serializability of  $H$ , and let  $S'$  be any sequential extension  $S'$  of  $S$  in accordance to Def. 12. Since  $T_i$  aborts in  $H$ , then it is not in  $S$ , and therefore it is replaced in  $S'$  by  $T_i^{gr}$ . Since, by construction,  $T_i^{gr}$  does not contain any write operation executions, there is no write operation execution writing  $v$  to  $x$  in  $S'$ . Since  $T_i$  released  $x$  early in  $H$  there is a transaction  $T_j$  in  $H$  that executes a read operation reading  $v$  from  $x$  and the same read operation is in  $S'$ . But since there is no write operation execution writing  $v$  to  $x$  in  $S'$ , no transaction containing a read operation execution reading  $v$  from  $x$  can be legal in  $S'$ . Thus,  $H$  is not live opaque, which is a contradiction.

Therefore  $H$  cannot be simultaneously live opaque and contain a transaction with early release that aborts.  $\square$

Like with VWC, live opacity does not allow transactions that release early to abort, which we consider too strict a condition.

### 3.6 Elastic Opacity

*Elastic opacity* is a safety property based on opacity, that was introduced to describe the safety guarantees of elastic transactions [13]. The property allows to relax the atomicity requirement of transactions to allow each of them to execute as a series of smaller transactions. An *elastic transaction*  $T_i$  is split into a sequence of subhistories called a *cut* denoted  $C_i(H)$ , where each subhistory represents a "subtransaction." In brief, a cut that contains more than one operation execution is *well-formed* if all subhistories are longer than one operation execution, all the write operations are in the same subhistory, and the first operation execution on any variable in every subhistory is not a write operation, except possibly in the first subhistory. A well-formed cut of some transaction  $T_i$  is consistent in some history  $H$ , if given any two operation executions  $op_i$  and  $op'_i$  on  $x$  in any subhistories of the cut, no transaction  $T_j$  ( $i \neq j$ ) executes a write operation  $op_j$  on  $x$  s.t.  $op_i <_H op_j <_H op'_i$ . In addition, given any two operation executions  $op_i$  and  $op'_i$  on  $x, y$  respectively, no two transactions  $T_k, T_l$  ( $l \neq i, k \neq i$ ) execute writes  $op_k$  on  $x$  and  $op_l$  on  $y$ , s.t.  $op_i <_H op_k <_H op'_i$  and  $op_i <_H op_l <_H op'_i$ . A *cutting function*  $f_C$  takes a history  $H$  as an argument and produces a new history  $H_f$  where for each transaction  $T_i \in H$  declared as elastic,  $T_i$  is replaced in  $H_f$  with the transactions resulting from the cut  $C_i(H)$  of  $T_i$ . If some transaction is committed (aborted) in  $H$ , then all transactions resulting from its cut are committed (aborted) in  $f_C(H)$ . Then, elastic opacity is defined as follows:

**Definition 13** (Elastic Opacity). *History  $H$  is elastic opaque iff there exists a cutting function  $f_C$  that replaces each elastic transaction  $T_i$  in  $H$  with its consistent cut  $C_i(H)$ , such that history  $f_C(H)$  is opaque.*

**Theorem 12.** *Elastic opacity supports early release.*

*Proof.* Let  $H$  be a transactional history with unique writes as shown in Fig. 6. Let  $T_i$  be an elastic transaction. Let  $C_i(H)$  be a cut of subhistory  $H|T_i$ , such that:

$$C_i(H) = \{[start_{i'} \rightarrow ok_{i'}, r_{i'}(y) \rightarrow 0, w_{i'}(x)1 \rightarrow ok_{i'}, tryC_{i'} \rightarrow C_{i'}], \\ [start_{i''} \rightarrow ok_{i''}, r_{i''}(x) \rightarrow 1, r_{i''}(y) \rightarrow 0, tryC_{i''} \rightarrow C_{i''}]\}.$$

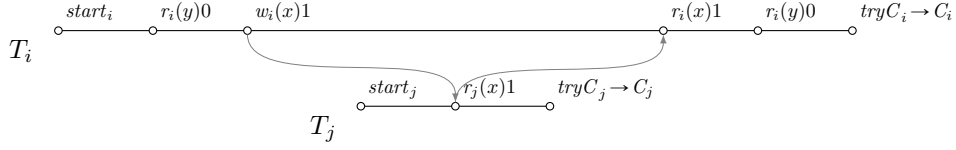


Figure 6: Elastic opaque history with early release.

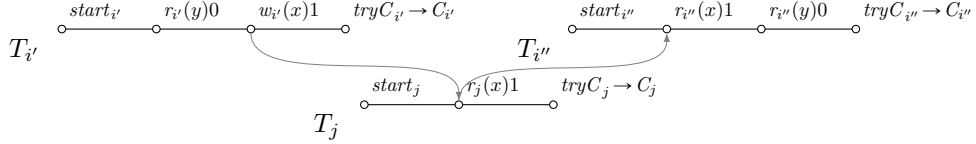


Figure 7: History after applying a cutting function.

All subhistories of  $C_i(H)$  are longer than one operation, all the writes are in the first subhistory, and no subhistory starts with a write, so  $C_i(H)$  is well-formed. Since there are no write operations outside of  $T_i$ , then it follows that  $C_i(H)$  is a consistent cut in  $H$ . Let  $f_C$  be any cutting function such that it cuts  $T_i$  according to  $C_i(H)$ , in which case  $f_C(H)$  is defined as in Fig. 7. Let  $S$  be a sequential history s.t.  $S = f_C(H)|T_{i'} \cdot f_C(H)|T_j \cdot f_C(H)|T_{i''}$ . Since  $T_{i'}$  precedes  $T_{i''}$  in  $S$  as well as in  $f_C(H)$ , and all other transactions are not real time ordered,  $S$  preserves the real time order of  $f_C(H)$ . Trivially, each transaction in  $S$  is legal in  $S$ . Thus,  $f_C(H)$  is opaque by Def. 8, and in effect  $H$  is elastic opaque by Def. 13. Since in  $H$  transaction  $T_j$  reads  $x$  from  $T_i$  while  $T_i$  is live, then, by Def. 2,  $T_i$  releases  $x$  early in  $x$ . Hence, since  $H$  is elastic opaque, elastic opacity supports early release, by Def. 3.  $\square$

**Theorem 13.** *Elastic opacity does not support overwriting.*

*Proof.* For the sake of contradiction, let us assume that there is an elastic opaque history  $H$  s.t. transaction  $T_i$  writes value  $v$  to some variable  $x$  and releases it early in  $H$ . Furthermore, let us assume that there is overwriting, so after some transaction  $T_j$  reads  $v$  from  $x$ ,  $T_i$  writes  $u$  to  $x$ . Since only elastic transactions can release early in elastic opaque histories, and  $T_i$  releases early,  $T_i$  is necessarily elastic. Thus, in any  $f_C(H)$   $T_i$  is replaced by a cut  $C_i(H)$ .

The two writes on  $x$  in  $T_i$  are either a) in two different subhistories in  $\mathcal{C}_H(i)$ , or b) in the same subhistory in  $\mathcal{C}_H(i)$ . Since the definition of a consistent cut requires all writes on a single variable are within one subhistory of the cut, then in case (a),  $\mathcal{C}_H(i)$  is inconsistent. Since by Def. 13 elastic opaque histories are created using consistent cuts, then  $H$  is not elastic opaque, which is a contradiction.

In the case of (b), let us say that both writes are in a subhistory that is converted into transaction  $T'_i$  in  $f_C(H)$ . Since  $T_i$  releases  $x$  early, then by Def. 2, there is a transaction  $T'_j$  in  $f_C(H)$  which executes a read on  $x$  reading the value written by  $T'_i$  in  $f_C(H)$ . Since we assume overwriting, the read operation on  $x$  in  $T'_j$  reads the value written by the first of the two writes in  $T'_i$  and does so before the other write on  $x$  is performed within  $\mathcal{C}_H(i)$ . Then, in any sequential history  $S$  equivalent to  $f_C(H)$  either  $T'_j <_S T'_i$  or  $T'_i <_S T'_j$ . In the former case  $T'_j$  in  $S$  is not legal in  $S$ , since the read on  $x$  that yields value  $v$  will not be preceded by any operation that writes  $v$  to  $x$  in any possible  $Vis(S, T'_j)$ . In the latter case

$T'_j$  in  $S$  is also not legal in  $S$ , since there will be a write operation writing  $u$  to  $x$  between the read on  $x$  that yields value  $v$  and any operation that writes  $v$  to  $x$  in  $Vis(S, T'_j)$ . Since  $T'_j$  in  $S$  is not legal in any  $S$  equivalent to  $f_C(H)$ , then, by Def. 7,  $f_C(H)$  is not final-state opaque, and hence, by Def. 8, not opaque. In effect, by Def. 13,  $H$  is not opaque, which is a contradiction.

Thus, there cannot be an elastic opaque history  $H$  with overwriting.  $\square$

**Theorem 14.** *Elastic opacity does not support early release aborting.*

*Proof.* For the sake of contradiction, let us assume that there is an elastic opaque history  $H$  s.t. transaction  $T_i$  releases some variable  $x$  early in  $H$  and aborts. Since  $T_i$  releases early then it writes  $v$  to  $x$ , and there is another  $T_j$  that executes a read on  $x$  that returns  $v$  before  $T_i$  aborts. Since only elastic transactions can release early in elastic opaque histories, and  $T_i$  releases early,  $T_i$  is necessarily elastic. If  $T_i$  aborts in  $H$ , then all of the transactions resulting from its cut  $C_i(H)$  in  $f_C(H)$  also abort (by construction of  $f_C(H)$ ). Therefore, for any sequential history  $S$  equivalent to  $f_C(H)$ , there is no subhistory  $H' \in C_i(H)$  s.t.  $H' \subseteq Vis(S, T_j)$ , and in effect the read operation in  $T_j$  on  $x$  reading  $v$  is not preceded by a write operation writing  $v$  to  $x$ . Therefore,  $Vis(S, T_j)$  is illegal, so  $T_j$  in  $S$  is not legal in  $S$ , and thus, by Def. 8  $f_C(H)$  is not opaque. Since  $f_C(H)$  is not opaque, then by Def. 13,  $H$  is not elastic opaque, which is a contradiction.  $\square$

Elastic opacity supports early release, but, since it does not guarantee serializability (as shown in [13]), we consider it to be a relatively weak property. This is contrary to our premise of finding a property that allows early release and provides stronger guarantees than serializability. Elastic transactions were proposed as an alternative to traditional transactions for implementing search structures, but we submit that the restrictions placed on the composition of elastic transactions and the need for transactions with early release to be non-aborting put an unnecessary burden on general-purpose TM. In particular, for a cut to be well-formed, it is necessary that all writes are executed in the same subtransaction, and that no subtransaction starts with a write, which severely limits how early release can be used and precludes scenarios that are nevertheless intuitively correct. In addition, elastic opacity requires that transactions which release early do not subsequently abort.

### 3.7 Database Properties

We follow the discussion of TM safety properties with a brief foray into database properties that deal with transaction consistency. Given that TM properties tend not to be very helpful when describing the behavior of early release, these consistency properties may be used to supplement that.

*Recoverability* is a database property defined as below (following [16]):

**Definition 14** (Recoverability). *History  $H$  is recoverable iff for any  $T_i, T_j \in H$ , s.t.  $i \neq j$  and  $T_j$  reads from  $T_i$ ,  $T_i$  commits in  $H$  before  $T_j$  commits.*

Recoverability does not make requirements about values read by transactions, so it necessarily supports early release, overwriting, and aborting early release. It also allows histories that are not even serializable. As such, it is too weak for application in TM. Recoverability can be combined with serializability to restrict the order on commits and aborts in serializable histories. The resulting consistency condition is therefore stronger

than serializability. However, it still allows unrestricted early release, overwriting, and aborting early release, and thus is not suitable for TM.

*Avoiding cascading aborts* (ACA) [7] is a database property defined as:

**Definition 15** (Avoiding Cascading Aborts). *History  $H$  Avoids Cascading Aborts iff for any  $T_i, T_j \in H$  s.t.  $i \neq j$  and  $T_j$  reads from  $T_i$ ,  $T_i$  commits before the read.*

As with recoverability, ACA restricts reading from live transactions. Therefore, ACA clearly removes all the scenarios encompassed by Def. 2. Since this is the only provision of ACA, the property forbids early release, without giving any additional guarantees. Hence, it also does not support overwriting nor aborting early release.

*Strictness* [7] is a database property defined as:

**Definition 16** (Strictness). *History  $H$  is strict iff for any  $T_i, T_j \in H$  ( $i \neq j$ ) and given any operation execution  $op_i = r_i(x) \rightarrow v$  or  $w_i(x)v' \rightarrow ok_i$  in  $H|T_i$ , and any operation execution  $op_j = w_j(x)v \rightarrow ok_j$  in  $H|T_j$ , if  $op_i$  follows  $op_j$ , then  $T_j$  commits or aborts before  $op_i$ .*

The definition unequivocally states that a transaction cannot read from another transaction, until the latter is committed or aborted. Thus, strictness precludes early release altogether. Hence, it also does not support overwriting nor aborting early release.

*Rigorousness* is defined (following [9]) as:

**Definition 17** (Rigorousness). *History  $H$  is rigorous iff it is strict and for any  $T_i, T_j \in H$  ( $i \neq j$ ) such that  $T_i$  writes to variable  $x$ , i.e.,  $op_i = w_i(x)v \rightarrow ok_i \in H|T_i$  after  $T_j$  reads  $x$ , then  $T_j$  commits or aborts before  $op_i$ .*

Since in [4] the authors demonstrate that rigorous histories are opaque, and since we show in Theorem 4 that opaque histories do not support early release, then neither does rigorousness. Hence, it also does not support overwriting nor aborting early release.

### 3.8 Discussion

The survey of properties shows that, while there are many safety properties for TM with a wide range of guarantees they provide, with respect to early release they fall into three basic groups.

The first group consists of properties that allow early release but do not prevent overwriting: serializability and recoverability. These properties do not control what can be seen by aborting transactions. As argued in [15], this is insufficient for TM in general, because operating on inconsistent state may lead to uncontrollable errors, whose consequences include crashing the process.

The second group consists of properties that preclude the dangerous situations allowed by the first group. This group includes opacity, TMS1, TMS2, ACA, strictness, and rigorousness. The properties in this group forbid early release altogether and obviously are not suited for TM systems that employ that mechanism.

The third group allows early release and precludes overwriting but also precludes aborting in transactions that release early. It includes live opacity, elastic opacity, and VWC. These properties seem to provide a reasonable middle ground between allowing early release and eliminating inconsistent views. However, these properties effectively require that transactions that release early become irrevocable. That is, once a live transaction is read from, it can never abort. The need to deal with irrevocable transactions is detrimental,

```

 $\mathcal{P}_1$ :  1 transaction { // spawns as  $T_1$ 
        2   x = 1;
        3   if (y > 0)
        4       x = x + y;
        5   y = x + 1;
        6 }

 $\mathcal{P}_2$ :  1 transaction { // spawns as  $T_2$ 
        2   y = y + 1;
        3 }

```

Figure 8: Transactional program with closing write.

because irrevocable transactions introduce additional complexity to a TM (see e.g., [37]). In addition, in applications like distributed computing, transaction aborts may be induced by external stimuli, so it can be completely impossible to prevent transactions from aborting [30]. Finally, the requirement to have transactions that release early eventually commit unnecessarily precludes some intuitively correct histories (see Fig. 3).

In summary, properties from the first group are not adequate for *any* TM and those from the second group do not allow any form of early release. The third group imposes an overstrict restriction that transactions which release early be irrevocable. None of the properties provide a satisfactory, strong safety property that could be used for a TM with early release, where aborts cannot be arbitrarily restricted. Therefore, a property expressing the guarantees of such systems is lacking. Hence, we introduce a property in Section 4 to fill this niche.

## 4 Last-use Opacity

We present *last-use opacity*, a new TM safety property that provides strong consistency guarantees and allows early release without compromising on the ability of transactions to abort. The property is based on the preliminary work in [32, 33].

The idea of last-use opacity hinges on identifying the *closing write* operation execution on a given variable in individual transactions. Informally, a closing write on some variable is such, that the transaction which executed it will not subsequently execute another write operation on the same variable in any *possible* extension of the history. What is possible is determined by the program that is being evaluated to create that history. Knowing the program, it is possible to infer (to an extent) what operations a particular transaction will execute. Hence, knowing the program, we can determine whether a particular operation on some variable is the last possible such operation on that variable within a given transaction. Thus, we can determine whether a given operation is the closing write operation in a transaction.

Take, for instance, the program in Fig. 8, where subprogram  $\mathcal{P}_1$  spawns transaction  $T_1$ , and  $\mathcal{P}_2$  spawns  $T_2$ . Let us assume that initially  $x$  and  $y$  are set to 0. Depending on the semantics of the TM, as these subprograms interweave during the execution, a number of histories can be produced. We can divide all of among them into two cases. In the first case  $T_2$  writes 1 to  $y$  in line 2 of  $\mathcal{P}_2$  and this value is then read by  $T_1$  in line 3 of  $\mathcal{P}_1$ . As a consequence,  $T_1$  will execute the write operation in line 4. The second case assumes that  $T_1$  reads 0 in line 3 of  $\mathcal{P}_1$  (e.g., because  $T_2$  executed line 2 much later). In this case,  $T_1$  will not execute the write operation in line 4. We can see, however, that in either of the above cases, once  $T_1$  executes the write to  $x$  on line 4, then no further writes to  $x$  will follow in  $T_1$  in any conceivable history. Thus, the write operation execution generated by line 4 of  $\mathcal{P}_1$  is

the closing write on  $x$  in  $T_1$ . On the other hand, the write operation execution generated by line 2 of  $\mathcal{P}_1$  is never the closing write on  $x$  in  $T_1$ , because there exists a conceivable history where another write operation execution will appear (i.e., once line 4 is evaluated). This is true even in the second of the cases because line 4 can be executed *in potentia*, even if it is not executed *de facto*.

Note that once any transaction  $T_i$  completes executing its closing write on some variable  $x$ , it is certain that no further modifications to that variable are intended by the programmer as part of  $T_i$ . This means, from the perspective of  $T_i$  (and assuming no other transaction modifies  $x$ ), that the state of  $x$  would be the same at the time of the closing write as if the transaction attempted to commit. Hence, with respect to  $x$ , we can treat  $T_i$  as if it had attempted to commit.

Last use opacity uses the concept of a closing write to dictate one transaction can read from another transaction. We give a formal definition in Section 4.1, but, in short, given any two transactions,  $T_i$  and  $T_j$ , last-use opacity allows  $T_i$  to read variable  $x$  from  $T_j$  if the latter is either committed or commit-pending, or, if  $T_j$  is live and it already executed its closing write on  $x$ . This has the benefit of allowing early release while excluding overwriting completely. However, last-use opacity does allow cascading aborts to occur. We discuss their implications in Section 4.3, as well as ways of mitigating them. That section also describes the guarantees given by last-use opacity.

## 4.1 Definition

First, we define the concept of a closing write to some variable by a particular transaction. We do this by first defining a closing write operation invocation, and then extend the definition to complete operation executions.

Given program  $\mathbb{P}$  and a set of processes  $\Pi$  executing  $\mathbb{P}$ , since different interleavings of  $\Pi$  cause an execution  $\mathcal{E}(\mathbb{P}, \Pi)$  to produce different histories, then let  $\mathbb{H}^{\mathbb{P}, \Pi}$  be the set of all possible histories that can be produced by  $\mathcal{E}(\mathbb{P}, \Pi)$ , i.e.,  $\mathbb{H}^{\mathbb{P}, \Pi}$  is the largest possible set s.t.  $\mathbb{H}^{\mathbb{P}, \Pi} = \{H \mid H \models \mathcal{E}(\mathbb{P}, \Pi)\}$ .

**Definition 18** (Closing Write Invocation). *Given a program  $\mathbb{P}$ , a set of processes  $\Pi$  executing  $\mathbb{P}$  and a history  $H$  s.t.  $H \models \mathcal{E}(\mathbb{P}, \Pi)$ , i.e.  $H \in \mathbb{H}^{\mathbb{P}, \Pi}$ , an invocation  $inv_i(w(x)v)$  is the closing write invocation on some variable  $x$  by transaction  $T_i$  in  $H$ , if for any history  $H' \in \mathbb{H}^{\mathbb{P}, \Pi}$  for which  $H$  is a prefix (i.e.,  $H' = H \cdot R$ ) there is no operation invocation  $inv_i(w(x)u)$  s.t.  $inv_i(w(x)v)$  precedes  $inv_i(w(x)u)$  in  $H'|T_i$ .*

**Definition 19** (Closing Write). *Given a program  $\mathbb{P}$ , a set of processes  $\Pi$  executing  $\mathbb{P}$  and a history  $H$  s.t.  $H \models \mathcal{E}(\mathbb{P}, \Pi)$ , an operation execution is the closing write on some variable  $x$  by transaction  $T_i$  in  $H$  if it comprises of an invocation and a response other than  $A_i$ , and the invocation is the closing write invocation on  $x$  by  $T_i$  in  $H$ .*

The *closing read invocation* and the *closing read* are defined analogously.

If a transaction executes its closing write on some variable, we say that the transaction *decided on  $x$* .

**Definition 20** (Transaction Decided on  $x$ ). *Given a program  $\mathbb{P}$ , a set of processes  $\Pi$  and a history  $H$  s.t.  $H \models \mathcal{E}(\mathbb{P}, \Pi)$ , we say transaction  $T_i \in H$  decided on variable  $x$  in  $H$  iff  $H|T_i$  contains a complete write operation execution  $w_i(x)v \rightarrow ok_i$  that is the closing write on  $x$ .*

Given some history  $H$ , let  $\hat{\mathbb{T}}^H$  be a set of transactions s.t.  $T_i \in \hat{\mathbb{T}}^H$  iff there is some variable  $x$  s.t.  $T_i$  decided on  $x$  in  $H$ .

Given any  $T_i \in H$ , a *decided transaction subhistory*, denoted  $H|T_i$ , is the longest subsequence of  $H|T_i$  s.t.:

- a)  $H|T_i$  contains  $start_i \rightarrow u$ , and
- b) for any variable  $x$ , if  $T_i$  decided on  $x$  in  $H$ , then  $H|T_i$  contains  $(H|T_i)|x$ .

In addition, a *decided transaction subhistory completion*, denoted  $H|T_i^\circ$ , is a sequence s.t.  $H|T_i^\circ = H|T_i \cdot [tryC_i \rightarrow C_i]$ .

Given a sequential history  $S$  s.t.  $S \equiv H$ ,  $LVis(S, T_i)$  is the longest subhistory of  $S$ , s.t. for each  $T_j \in S$ :

- a) if  $i = j$  or  $T_j$  is committed in  $S$  and  $T_j <_S T_i$ , then  $S|T_j \subseteq LVis(S, T_i)$ ,
- b) if  $T_j$  is not committed in  $S$  but  $T_j \in \hat{\mathbb{T}}^H$  and  $T_j <_S T_i$ , and it is not true that  $T_j <_H T_i$ , then either  $S|T_j \subseteq LVis(S, T_i)$  or not.

Given a sequential history  $S$  and a transaction  $T_i \in S$ , we then say that transaction  $T_i$  is *last-use legal in  $S$*  if  $LVis(S, T_i)$  is legal. Note that if  $S$  is legal, then it is also last-use legal (see appendix for proof).

**Definition 21** (Final-state Last-use Opacity). *A finite history  $H$  is final-state last-use opaque if, and only if, there exists a sequential history  $S$  equivalent to any completion of  $H$  s.t.,*

- a)  $S$  preserves the real-time order of  $H$ ,
- b) every transaction in  $S$  that is committed in  $S$  is legal in  $S$ ,
- c) every transaction in  $S$  that is not committed in  $S$  is last-use legal in  $S$ .

**Definition 22** (Last-use Opacity). *A history  $H$  is last-use opaque if, and only if, every finite prefix of  $H$  is final-state last-use opaque.*

**Theorem 15.** *Last-use opacity is a safety property.*

*Proof.* By Def. 22, last-use opacity is trivially prefix-closed.

Given  $H_L$  that is an infinite limit of any sequence of finite histories  $H_0, H_1, \dots$ , s.t every  $H_h$  in the sequence is last-use-opaque and every  $H_h$  is a prefix of  $H_{h+1}$ , since each prefix  $H_h$  of  $H_L$  is last-use-opaque, then, by extension, every prefix  $H_h$  of  $H_L$  is also final-state last-use opaque, so, by Def. 22,  $H_L$  is last-use-opaque. Hence, last-use opacity is limit-closed.

Since last-use opacity is both prefix-closed and limit-closed, then, by Def. 1, it is a safety property.  $\square$

## 4.2 Examples

In order to aid understanding of the property we present examples of last-use opaque histories in Fig. 9–12. These are contrasted by examples of histories that are not last-use opaque in Fig. 14–17. We discuss the examples below and prove them in the appendix.

The example in Fig. 9 shows  $T_i$  executing a write on  $x$  once and releasing  $x$  early to  $T_j$ . We assume that the program generating the history is such, that the write operation executed by  $T_i$  is the closing write operation execution on  $x$ . The history is intuitively

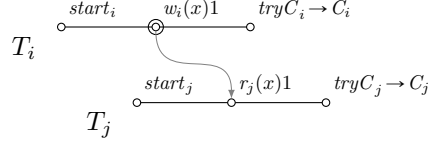


Figure 9: Example satisfying last-use opacity: early release. We mark a closing write operation execution in some history in the diagram as  $\ominus$ . Note that an operation can be the ultimate operation execution in some transaction, but still not fit the definition of a closing operation execution.

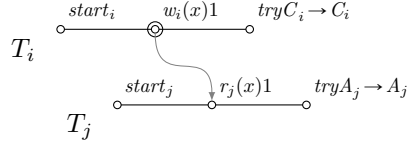


Figure 10: Example satisfying last-use opacity: early release to an aborting transaction.

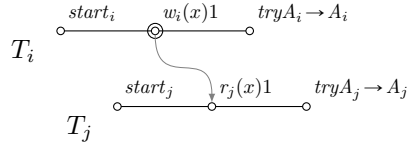


Figure 11: Example satisfying last-use opacity: early release between two aborting transactions.

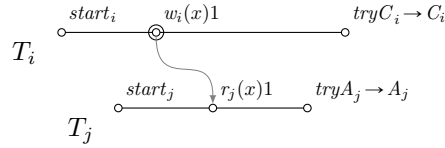


Figure 12: Example satisfying last-use opacity: early release to a prematurely aborting transaction.

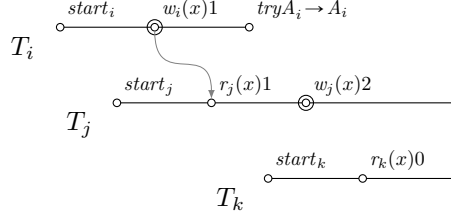


Figure 13: Example satisfying last-use opacity: freedom to read from or ignore an aborted transaction.

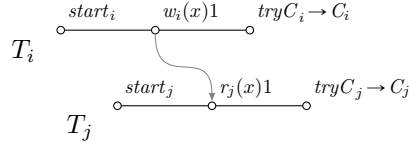


Figure 14: Example breaking last-use opacity: early release before closing write operation execution.

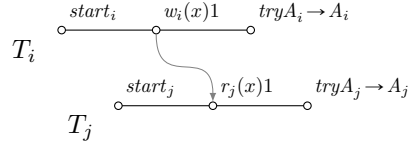


Figure 15: Example breaking last-use opacity: early release between two aborting transactions before closing write operation execution.

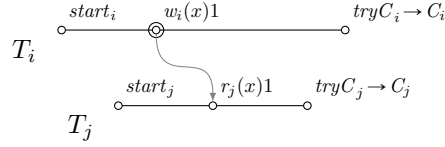


Figure 16: Example breaking last-use opacity: commit order not respected.

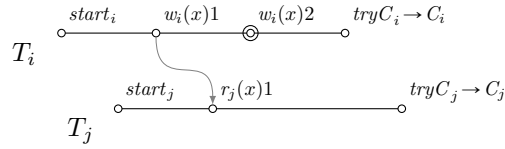


Figure 17: Example breaking last-use opacity: early release with overwriting.

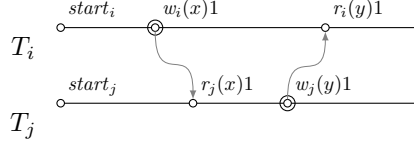


Figure 18: Example breaking last-use opacity: dependency cycle.

correct, since both transactions commit, and  $T_j$  reads a value written by  $T_i$ . On the formal side, since both transactions are committed in this history, the equivalent sequential history would consist of all the events in  $T_i$  followed by the events in  $T_j$  and both transactions would be legal, since  $T_i$  writes a legal value to  $x$  and  $T_j$  reads the last value written by  $T_i$  to  $x$ . Thus, the history is final-state last-use opaque.

Since last-use opacity requires prefix closedness, then all prefixes of the history in Fig. 9 also need to be final-state last-use opaque. We present only two of the interesting prefixes, since the remainder are either similar or trivial. The first interesting prefix is created by removing the commit operation execution from  $T_j$ , which means  $T_j$  is aborted in any completion of the history. We show such a completion in Fig. 10. Still,  $T_i$  writes a legal value to  $x$  and  $T_j$  reads the last value written by  $T_i$  to  $x$ , so that prefix is also final-state last-use opaque. Another interesting prefix is created by removing the commit operation executions from both transactions. Then, in the completion of the history both transactions are aborted, as in Fig. 11. Then, in an equivalent sequential history  $T_j$  would read a value written by an aborted transaction. In order to show legality of a committed transaction, we use the subhistory denoted  $Vis$ , which does not contain any transactions that were not committed in the history from which it was derived. Thus, if  $T_j$  were committed, it would not be legal, since its  $Vis$  would not contain a write operation execution writing the value the transaction actually read. However, since  $T_j$  is aborted, the definition of final-state last-use opacity only requires that  $LVis$  rather than  $Vis$  be legal, and  $LVis$  can contain operation executions on particular variables from an aborted transaction under the condition that the transaction already executed its closing write on the variables in question. Since, in the example  $T_i$  executed its closing write on  $x$ , then this write will be included in  $LVis$  for  $T_j$ , so  $T_j$  will be last-use legal. In consequence the prefix is also final-state last-use opaque. Indeed, all prefixes of example Fig. 9 are final-state last-use opaque, so the example is last-use opaque, and, by extension, so are the examples in Fig. 10 and Fig. 11.

Contrast the example in Fig. 9 with the one in Fig. 14. The histories presented in both are identical, with the exception that the write operation in Fig. 9 is considered to be the closing operation execution, while in Fig. 14 it is not. The difference would stem from differences in the programs that produced these histories. For instance, the program producing the history in Fig. 14 could conditionally execute another operation on  $x$ , so, even though that condition was not met in this history, the potential of another write on  $x$  means that the existing write cannot be considered a closing write operation execution. The consequence of this is that while the example itself is final-state last-use opaque, one of its prefixes is not, so the history is not last-use opaque. The offending prefix is created by removing commit operations in both transactions, so both transactions would abort in any completion, as in Fig. 15. Here, since  $T_i$  does not execute the closing write operation on  $x$ , then the write operation would not be included in  $LVis$  for  $T_j$ , so the value read by  $T_j$  could not be justified. Thus,  $T_j$  is not legal in that history, and, therefore, the history

in Fig. 15 is not final-state last-use opaque (so also not last-use opaque). Fig. 15 represents the completion of a prefix of the history in Fig. 14, so Fig. 15 not being final-state last-use opaque, means that Fig. 14 is not last-use opaque.

The examples in Fig. 12 and Fig. 16, show that recoverability is required, i.e., transactions must commit in order. Last-use opacity of the example in Fig. 12 is analogous to the one in Fig. 10, since their equivalent sequential histories are identical, as are the sequential histories equivalent to their prefixes. Furthermore, intuitively, if  $T_j$  reads a value of a variable released early by  $T_i$  and aborts before  $T_i$  commits, this is correct behavior. On the other hand, the history in Fig. 16 is not last-use opaque, even though it is final-state last-use opaque (by analogy to Fig. 9). However, a prefix of the history where the commit operation execution is removed from  $T_i$  is not final-state last-use opaque. This is because a completion will require that  $T_i$  be aborted, the operations executed by  $T_i$  are not going to be included in any  $Vis$ . Since  $T_j$  is committed, then its  $Vis$  must be legal, but it is not, because the read operation reading 1 will not be preceded by any writes in  $Vis$ . Since the prefix contains an illegal transaction, then it is not final-state last-use opaque, and thus, the history in Fig. 16 is not last-use opaque.

The example in Fig. 13 shows that a transaction is allowed to read from a transaction that eventually aborts, or ignore that transaction, because of the freedom left within the definition of  $LVis$ . I.e. transactions  $T_j$  is concurrent to  $T_i$ , but  $T_k$  follows  $T_i$  in real time.  $T_i$  executes a closing write on  $x$ , so  $T_j$  is allowed to include the write operation on  $x$  in its  $LVis$ . Since  $T_j$  sees the value written to  $x$  by that write,  $T_j$  includes the write in  $LVis$ . On the other hand,  $T_k$  cannot include  $T_i$ 's write in  $LVis$ , since it aborted before  $T_k$  even started, so the write should not be visible to  $T_k$ . On the other hand  $T_k$  is allowed to include  $T_j$  in its  $LVis$ .  $T_k$  should not do so, however, since it ignores  $T_j$  as well as  $T_i$  (which makes sense as  $T_j$  is doomed to abort). Hence  $T_k$  reads the value of  $x$  to be 0. If  $T_j$  is included in  $T_k$ 's  $LVis$ , reading 0 would be incorrect. Hence, the definition of  $LVis$  allows  $T_j$  to be arbitrarily excluded. In effect all three transactions are correct (so long as  $T_j$  does not eventually commit).

Fig. 17 shows an example of overwriting, which is not last-use opaque, since there is no equivalent sequential history where the write operation in  $T_i$  writing 1 to  $x$  would precede the read operation in  $T_j$  reading 1 from  $x$  without the other write operation writing 2 to  $x$  also preceding the read. Thus, in all cases  $T_j$  is not legal, and the history is neither final-state last-use opaque, nor last-use opaque.

Finally, Fig. 18 shows an example of a cyclic dependency, where  $T_j$  reads  $x$  from  $T_i$ , and subsequently  $T_i$  reads  $y$  from  $T_j$ . Both writes in the history are closing writes. This example has unfinished transactions, which are thus aborted in any possible completion of this history. There are two possible sequential histories equivalent to that completion: one where  $T_i$  precedes  $T_j$  and one where  $T_j$  precedes  $T_i$ . In the former case,  $LVis$  of  $T_i$  does not contain any operations from  $T_j$ , because  $T_j$  follows  $T_i$ . Thus, there is no write operation on  $y$  preceding a read on  $y$  returning 1 in  $T_i$ 's  $LVis$ , which does not conform to the sequential specification, so  $T_i$ 's  $LVis$  is not legal. Hence,  $T_i$  is not legal in that scenario. The former case is analogous:  $T_j$ 's  $LVis$  will not contain a write operation from  $T_i$ , because  $T_i$  follows  $T_j$ . Therefore  $T_j$ 's  $LVis$  contains a read on  $x$  that returns 1, which is not preceded by any write on  $x$ , which causes the sequence not to conform to the sequential specification and renders the transaction not legal. Since either case contains a transaction that is not legal, then that history is not final-state last-use opaque, and therefore not last-use opaque.

### 4.3 Guarantees

Last-use opacity gives the programmer the following guarantees:

**Serializability** If a transaction commits, then the value it reads can be explained by operations executed by preceding or concurrent transactions. This guarantees that a transaction that views inconsistent state will not commit.

**Lemma 1.** *Every last-use-opaque history is serializable.*

We provide the proof for Lemma 10.

**Real-time Order** Successive transactions will not be rearranged to fit serializability, so a correct history will agree with an external clock, or an external order of events.

**Lemma 2.** *Every last-use-opaque history preserves real-time order.*

*Proof.* Trivially from Def. 22 and Def. 21a. □

**Recoverability** If one transaction reads from another transaction, the former will commit only after the latter commits. This guarantees that transactions commit in order.

**Lemma 3.** *Every last-use-opaque history is recoverable.*

We provide the proof in Appendix B.

**Precluding Overwriting** If transaction  $T_i$  reads the value of some variable written by transaction  $T_j$ , then  $T_j$  will never subsequently modify that variable.

**Lemma 4.** *Last-use opacity does not support overwriting.*

*Proof.* For the sake of contradiction let us assume that there exists  $H$  that is a last-use-opaque history with overwriting, i.e. (from Def. 4) there are transaction  $T_i$  and  $T_j$  s.t.:

- a)  $T_i$  releases some variable  $x$  early,
- b)  $H|T_i$  contains  $w_i(x)v \rightarrow ok_i$  and  $w_i(x)v' \rightarrow ok_i$ , s.t. the former precedes the latter in  $H|T_i$ ,
- c)  $H|T_j$  contains  $r_j(x) \rightarrow v$  that precedes  $w_i(x)v' \rightarrow ok_i$  in  $H$ .

Since  $H$  is opaque, then there is a completion  $C = Compl(H)$  and a sequential history  $S$  s.t.  $S \equiv H$ ,  $S$  preserves the real-time order of  $H$ , and both  $T_i$  and  $T_j$  in  $S$  are legal in  $S$ . In  $S$ , either  $T_i <_S T_j$  or  $T_j <_S T_i$ . In either case, any  $Vis(S, T_j)$  or  $LVis(S, T_j)$  by their definitions will contain either the sequence of both  $w_i(x)v \rightarrow ok_i$  and  $w_i(x)v' \rightarrow ok_i$  or neither of those write operation executions. In either case,  $r_j(x) \rightarrow v$  will not be directly preceded by  $w_i(x)v \rightarrow ok_i$  among operations on  $x$  in either  $Vis(S, T_j)$  or  $LVis(S, T_j)$ . Therefore,  $T_j$  in  $S$  cannot be legal in  $S$ , which is a contradiction. □

```

1 // invariant:  $x \geq 0$ 
2 transaction {
3    $x = y - 1$ ;
4   if ( $x < 0$ )
5     abort();
6 }

```

Figure 19: Abort example.

```

1 // invariant:  $x \geq 0$ 
2 transaction {
3    $*\_array + x$ ;
4 }

```

Figure 20: Memory error example.

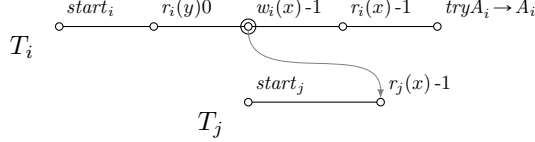


Figure 21: Last-use opaque history with inconsistent view.

**Aborting Early Release** A transaction can release some variable early and subsequently abort.

**Lemma 5.** *Last-use opacity supports aborting early release.*

*Proof.* Let  $H$  be the history depicted in Fig. 11. Here,  $T_i$  releases  $x$  early to  $T_j$  and subsequently aborts, which satisfies Def. 5. Since  $T_i$  and  $T_j$  are both aborted in  $H$ ,  $H$  has a completion  $C = \text{Compl}(H) = H$ . Let  $S$  be a sequential history s.t.  $S = H|T_i \cdot H|T_j$ .  $S$  vacuously preserves the real-time order of  $H$  and trivially  $S \equiv H$ . Transaction  $T_i$  in  $S$  is last-use legal in  $S$ , because  $LVis(S, T_i) = H|T_i$  whose operations on  $x$  are limited to a single write operation execution is within the sequential specification of  $x$ . Transaction  $T_j$  in  $S$  is also last-use legal in  $S$ , since  $LVis(S, T_j) = H|T_i \cdot H|T_j$  whose operations on  $x$  consist of  $w_i(x)v \rightarrow ok_i$  followed by  $r_j(x) \rightarrow v$  is also within the sequential specification of  $x$ . Since both  $T_i$  and  $T_j$  in  $S$  are last-use legal in  $S$ ,  $H$  is final-state last-use opaque. All prefixes of  $H$  are trivially also final-state last-use opaque (since either their completion is the same as  $H$ 's, they contain only a single write operation execution on  $x$ , or contain no operation executions on variables), so  $H$  is last-use opaque.  $\square$

**Exclusive Access** Any transaction has effectively exclusive access to any variable it accesses, at minimum, from the first to the final modification it performs, regardless of whether it eventually commits or aborts.

**Lemma 6.** *Any transaction in any last-use-opaque history has exclusive access to any variable between first and last access to.*

*Proof.* From Lemma 1 and Lemma 4.  $\square$

However, last-use opacity does not preclude transactions from aborting after releasing a variable early. As a consequence there may be instances of cascading aborts, which have varying implications on consistency depending on whether the TM model allows transactions to abort programmatically. We distinguish three cases of models and discuss them below.

**Only Forced Aborts** Let us assume that transactions cannot arbitrarily abort, but only do so as a result of receiving an abort response to invoking a read or write operation, or while attempting to commit. In other words, there is no *tryA* operation in the transactional API. In that case, since overwriting is not allowed, the transaction never reveals intermediate values of variables to other transactions. This means, that if a transaction released a variable early, then the programmer did not intend to change the value of that variable. So, if the transaction eventually committed, the value of the variable would have been the same. So, if the transaction is eventually forced to abort rather than committing, the value of any variable released early would be the same regardless of whether the transaction committed or aborted. Therefore, we can consider the inconsistent state to be safe. In other words, if the variable caused an error to occur, the error would be caused regardless of whether the transaction finally aborts or commits. Thus, we can say that with this set of assumptions, the programmer is guaranteed that none of the inconsistent views will cause unexpected behavior, even if cascading aborts are possible. Note that the use of this model is not uncommon (see eg. [13, 2, 3]).

**Programmer-initiated Aborts** Alternatively, let us assume that transactions can arbitrarily abort (in addition to forced aborts as described above) by executing the operation *tryA* as a result of some instruction in the program. In that case it is possible to imagine programs that use the abort instruction to cancel transaction due to the "business logic" of the program. Therefore a programmer explicitly specifies that the value of a variable is different depending on whether the transaction finally commits or not. An example of such a program is given in Fig. 19. Here, the programmer enforced an invariant that the value of  $x$  should never be less than zero. If the invariant is not fulfilled, the transaction aborts. However, writing a value to  $x$  that breaks the invariant is the closing write operation execution for this program, so it is possible that another transaction reads the value of  $x$  before the transaction aborts. If the transaction that reads  $x$  is like the one in Fig. 20, where  $x$  is used to index an array via pointer arithmetic, a memory error is possible. Nevertheless, the history from Fig. 21 that corresponds to a problematic execution of these two transactions is clearly allowed by last-use opacity (assuming that the domain of  $x$  is  $\mathbb{Z}$ ). Thus, if the abort operation is available to the programmer the guarantee that inconsistent views will not lead to unexpected effects is lost. Therefore it is up to the programmer to use aborts wisely or to prevent inconsistent views from causing problems, by prechecking invariants at the outset of a transaction, or maintaining invariants also within a transaction (in a similar way as with monitor invariants). Alternatively, a mechanism can be built into the TM that prevents specific transactions at risk from reading variables that were released early, while other transactions are allowed to do so. However, if these workarounds are not satisfactory, we present a stronger variant of last-use opacity in Section 4.5 that deals specifically with this model and eliminates its inconsistent views.

**Arbitrary Aborts** We present a third alternative to aborts in transactions: a compromise between only forced aborts and programmer-initiated aborts. This option assumes that the *tryA* operation is not available to the programmer, so it cannot be used to implement business logic. However, we allow the TM system to somehow inject *tryA* operations in the code in response to external stimuli, such as crashes or exceptions and use aborts as a fault tolerance mechanism. However, since the programmer cannot use the operation, the programs must be coded as in the forced aborts case, and therefore the same guarantees are given.

$\mathbb{H}_{tms2}$	$\subset$	$\mathbb{H}_{lop}$	$\mathbb{H}_{tms1}$	$\not\subset$	$\mathbb{H}_{lop}$	$\wedge$	$\mathbb{H}_{tms1}$	$\not\supset$	$\mathbb{H}_{lop}$
$\mathbb{H}_{op}$	$\subset$	$\mathbb{H}_{lop}$	$\mathbb{H}_{eop}$	$\not\subset$	$\mathbb{H}_{lop}$	$\wedge$	$\mathbb{H}_{eop}$	$\not\supset$	$\mathbb{H}_{lop}$
$\mathbb{H}_{rig}$	$\subset$	$\mathbb{H}_{lop}$	$\mathbb{H}_{str}$	$\not\subset$	$\mathbb{H}_{lop}$	$\wedge$	$\mathbb{H}_{str}$	$\not\supset$	$\mathbb{H}_{lop}$
$\mathbb{H}_{lvop}$	$\subset$	$\mathbb{H}_{lop}$	$\mathbb{H}_{vwc}$	$\not\subset$	$\mathbb{H}_{lop}$	$\wedge$	$\mathbb{H}_{vwc}$	$\not\supset$	$\mathbb{H}_{lop}$
$\mathbb{H}_{rec}$	$\supset$	$\mathbb{H}_{lop}$	$\mathbb{H}_{aca}$	$\not\subset$	$\mathbb{H}_{lop}$	$\wedge$	$\mathbb{H}_{aca}$	$\not\supset$	$\mathbb{H}_{lop}$
$\mathbb{H}_{ser}$	$\supset$	$\mathbb{H}_{lop}$							

Figure 22: Last-use opaque histories  $\mathbb{H}_{lop}$  in relation to: TMS2 and TMS1 histories  $\mathbb{H}_{tms2}$ ,  $\mathbb{H}_{tms1}$ , opaque histories  $\mathbb{H}_{op}$ , elastic opaque histories  $\mathbb{H}_{eop}$ , rigorous histories  $\mathbb{H}_{rig}$ , strict histories  $\mathbb{H}_{str}$ , live opaque histories  $\mathbb{H}_{lvop}$ , virtual world consistent histories  $\mathbb{H}_{vwc}$ , recoverable histories  $\mathbb{H}_{rec}$ , histories avoiding cascading abort  $\mathbb{H}_{aca}$ , and serializable histories  $\mathbb{H}_{ser}$ .

#### 4.4 Last-use Opacity in Context

We compare last-use opacity with other safety properties with respect to their relative strength. Given two properties  $\mathfrak{P}_1$  and  $\mathfrak{P}_2$  and the set of all histories that satisfy each property  $\mathbb{H}_1$ ,  $\mathbb{H}_2$ , respectively.  $\mathfrak{P}_1$  is stronger than  $\mathfrak{P}_2$  if  $\mathbb{H}_1 \subset \mathbb{H}_2$  (so  $\mathfrak{P}_2$  is weaker than  $\mathfrak{P}_1$ ). If neither  $\mathbb{H}_1 \subset \mathbb{H}_2$  nor  $\mathbb{H}_1 \supset \mathbb{H}_2$ , then the properties are incomparable.

We present the result of the comparison in Fig. 22. We describe the comparison with opacity and serializability in particular below, and provide proofs for the comparison of the remaining properties in the appendix.

Opacity is strictly stronger than last-use opacity.

**Lemma 7.** *For any history  $S$  and transaction  $T_i \in S$ , if  $Vis(S, T_i)$  is legal, then  $LVis(S, T_i)$  is legal.*

*sketch.* By definition of  $Vis(S, T_i)$ , if operation  $op \in Vis(S, T_i)$ , then  $op \in Vis(S, T_i)$  only if  $op \in H|T_j$  and either  $i = j$  or  $T_j <_S T_i$  and  $T_j$  is committed. By definition of  $LVis(S, T_i)$ , given transactions  $T_i, T_j$  and operation  $op \in S|T_j$ , if  $i = j$  or  $T_j <_S T_i$  and  $T_j$  is committed, then  $S|T_j \subseteq LVis(S, T_i)$ . Therefore  $LVis(S, T_i) \equiv Vis(S, T_i)$ . Since  $Vis(S, T_i)$  and  $LVis(S, T_i)$  preserve the order of operations in  $S$ , then  $LVis(S, T_i) = Vis(S, T_i)$ . Hence, if  $Vis(S, T_i)$  is legal, then  $LVis(S, T_i)$  is legal.  $\square$

**Lemma 8.** *Any final-state last-use opaque history  $H$  is final-state last-use-opaque.*

*sketch.* From Def. 7, for any final-state opaque history  $H$ , there is a sequential history  $S \equiv Compl(H)$  s.t.  $S$  preserves the real time order of  $H$  and every transaction  $T_i$  in  $S$  is legal in  $S$ . Thus, for every transaction  $T_i$  in  $S$   $Vis(S, T_i)$  is legal. From the definition of completion, any  $T_i$  is either committed or aborted in  $Compl(H)$  and therefore likewise completed or aborted in  $S$ . If  $T_i$  is committed in  $S$ , then it is legal in  $S$ , so  $Vis(S, T_i)$  is legal, and therefore  $T_i$  is last-use legal in  $S$ . If  $T_i$  is aborted in  $S$ , then it is legal in  $S$ , so  $Vis(S, T_i)$  is legal, and therefore, from Lemma 7,  $LVis(S, T_i)$  is also legal, so  $T_i$  is last-use legal in  $S$ . Given that all transactions in  $S$  are last-use legal in  $S$ , then, from Def. 21,  $H$  is final-state last-use opaque.  $\square$

**Lemma 9.** *Any opaque history  $H$  is last-use-opaque.*

*Proof.* If  $H$  is opaque, then, from Def. 8, any prefix  $P$  of  $H$  is final-state opaque. Since any prefix  $P$  of  $H$  is final-state opaque, then, from Lemma 8, any  $P$  is also final-state last-use opaque. Then, by Def. 22  $H$  is last-use opaque.  $\square$

Last-use opacity is strictly stronger than serializability.

**Lemma 10.** *Any last-use-opaque history  $H$  is serializable.*

*Proof.* For the sake of contradiction let us assume that  $H$  is last-use-opaque and not serializable. Since  $H$  is last-use-opaque, then from Def. 22  $H$  is also final-state last-use opaque. Then, from Def. 21 there exists a completion  $H_C = \text{Compl}(H)$  such that there is a sequential history  $S$  s.t.  $S \equiv H_C$ ,  $S$  preserves the real-time order of  $H_C$ , and any committed transaction in  $S$  is legal in  $S$ . However, since  $H$  is not serializable, then from Def. 6 there does not exist a completion  $H_C = \text{Compl}(H)$  such that there is a sequential history  $S$  s.t.  $S \equiv H_C$ , and any committed transaction in  $S$  is legal in  $S$ . This contradicts the previous statement.  $\square$

## 4.5 Last-use Opacity Variant for the Programmer Initiated Abort Model

Even though last-use opacity prevents inconsistent views in the only forced aborts and arbitrary aborts models, it does not prevent inconsistent views in the programmer initiated aborts model. Hence, we present a variant of last-use opacity called  $\beta$ -last-use opacity ( $\beta$ lu opacity) that extends the definition of a closing write operation to take *tryA* operations into account, as if it was an operation that modifies a given variable.

**Definition 23** ( $\beta$ -Closing Write Invocation). *Given a program  $\mathbb{P}$ , a set of processes  $\Pi$  executing  $\mathbb{P}$  and a history  $H$  s.t.  $H \models \mathcal{E}(\mathbb{P}, \Pi)$ , i.e.  $H \in \mathbb{H}^{\mathbb{P}, \Pi}$ , an invocation  $\text{inv}_i(w(x)v)$  is the closing write invocation on some variable  $x$  by transaction  $T_i$  in  $H$ , if for any history  $H' \in \mathbb{H}^{\mathbb{P}, \Pi}$  for which  $H$  is a prefix (i.e.,  $H' = H \cdot R$ ) there is no operation invocation  $\text{inv}'$  s.t.  $\text{inv}_i(w(x)v)$  precedes  $\text{inv}'$  in  $H'|T_i$  where (a)  $\text{inv}' = \text{inv}_i(w(x)u)$  or (b)  $\text{inv}' = \text{inv}_i(\text{tryA})$ .*

The remainder of the definitions of  $\beta$ lu opacity are formed by analogy to their counterparts in last-use opacity. We only summarize them here and give their full versions in the appendix.

The definition of a  $\beta$ -closing write operation execution is analogous to that of closing write operation execution Def. 19. The  $\beta$ -closing write is used instead of the closing write to define a transaction  $\beta$ -decided on  $x$  in analogy to Def. 20. Then, that definition is used to define  $\hat{T}_\beta^H$ ,  $H|^\beta T_j$ , and  $H|^\beta T_j$  by analogy to  $\hat{T}^H$ ,  $H|T_j$  and  $H|T_j$ . Next, those definitions are used to define  $\beta LVis$  by analogy to  $LVis$ . Finally, we say a transaction  $T_i$  is  $\beta$ -last-use legal in some sequential history  $S$  if  $\beta LVis(S, T_i)$  is legal. This allows us to define  $\beta$ lu opacity as follows.

**Definition 24** (Final-state  $\beta$ -Last-use Opacity). *A finite history  $H$  is final-state  $\beta$ -last-use opaque if, and only if, there exists a sequential history  $S$  equivalent to any completion of  $H$  s.t.,*

- a)  $S$  preserves the real-time order of  $H$ ,
- b) every transaction in  $S$  that is committed in  $S$  is legal in  $S$ ,
- c) every transaction in  $S$  that is not committed in  $S$  is  $\beta$ -last-use legal in  $S$ .

**Definition 25** ( $\beta$ -Last-use Opacity). *A history  $H$  is  $\beta$ -last-use opaque if, and only if, every finite prefix of  $H$  is final-state  $\beta$ -last-use opaque.*

In this variant of last use opacity a transaction is not allowed to release a variable early if it is possible that the transaction may execute a voluntary abort. In effect,  $\beta$ lu opacity precludes inconsistent views in the programmer initiated abort model.

The  $\beta$ lu opacity property is trivially equivalent to last-use opacity in the only forced abort model (because there are no *tryA* operations in that model), but it is stronger than last-use opacity in the arbitrary abort model to the point of being overstrict.

The  $\beta$ lu opacity property is strictly stronger than last-use opacity in the arbitrary abort model, but it is too strong to be applicable to systems with early release. In the first place, even though the histories that are excluded by  $\beta$ lu opacity contain inconsistent views, these are harmless, because as we argue in Section 4.3, transactions always release variables with “final” values. These final values cannot be reverted by a programmer-initiated abort, so if the programmer sets up a closing write to a variable in a transaction, the value that was written was expected to both remain unchanged and be committed. Hence, it is acceptable for these values to be read by other transactions, even before the original transaction commits.

Secondly, the arbitrary abort model specifies that the TM system can inject a *tryA* operation into the transactional code to respond to some outside stimuli, such as crashes. Such events are unpredictable, so it may be possible for any transaction to abort at any time. Hence, it is necessary to assume that a *tryA* operation can be produced as the next operation invocation in any transaction at any time. In effect, as the definition of  $\beta$ lu opacity does not allow a transaction to release a variable early if a *tryA* is possible in the future,  $\beta$ lu opacity actually prevents early release altogether in the arbitrary abort model.

In summary,  $\beta$ lu opacity is a useful variant of last-use opacity to exclude inconsistent views in the programmer initiated abort model (if workarounds suggested in Section 4.3 are insufficient solutions). However  $\beta$ lu opacity is too strict for TMs operating in the arbitrary abort model, where it prevents early release altogether. For that reason, last-use opacity remains our focus.

## 5 Supremum Versioning Algorithm

In this section we discuss the Supremum Versioning Algorithm (SVA), a pessimistic concurrency control algorithm with early release and rollback support, and demonstrate that it satisfies last-use opacity. SVA in its current form was introduced in [34, 31], although the presentation here gives a more complete description. It builds on our rollback-free variant in [40, 38, 39] as well as an earlier version in [30]. Even though the current implementation of SVA as part of Atomic RMI is distributed, the concurrency control algorithm itself was created for multiprocessor TMs and is applicable to both distributed as well as multiprocessor systems.

The main aspect of SVA is the ability to release variables early. The early release mechanism in SVA is based on *a priori* knowledge about the maximum number of accesses that a transaction can perform on particular variables. We explored various methods of obtaining satisfactory upper bounds, including static analysis [29] and static typing [38]. A transaction that knows it performed exactly as many operations on some variable as the upper bound allows may then release that variable. SVA does not require the upper bounds to be precise, and can handle situation when they are either too great (some variables are not released early) or too low (transactions are aborted). Since SVA does not distinguish between reads and writes when releasing, but instead treats all accesses uniformly, we will

```

1 procedure start(Transaction  $T_i$ ) {
2   for ( $x : ASet(T_i)$  sorted by  $<_{lk}$ )
3     lock  $lk(x)$ 
4   for ( $x : ASet(T_i)$  in parallel) {
5      $gv(x) \leftarrow gv(x) + 1$ 
6      $pv_i(x) \leftarrow gv(x)$ 
7   }
8   for ( $x : ASet(T_i)$  sorted by  $<_{lk}$ )
9     unlock  $lk(x)$ 
10 }

12 procedure access(Transaction  $T_i$ , Var  $x$ ) {
13   wait until  $pv_i(x) - 1 = lv(x)$ 
14   checkpoint( $T_i$ ,  $x$ )
15   if ( $rv_i(x) \neq cv(x)$ )
16     abort( $T_i$ ) and exit
17   execute read or write
18    $ac_i(x) \leftarrow ac_i(x) + 1$ 
19   if ( $ac_i(x) = \text{supr}_i(x)$ ) {
20      $cv(x) \leftarrow pv_i(x)$ 
21      $lv(x) \leftarrow pv_i(x)$ 
22   }
23 }

25 procedure abort(Transaction  $T_i$ ) {
26   for ( $x : ASet(T_i)$  in parallel) {
27     wait until  $pv_i(x) - 1 = lt_v(x)$ 
28     dismiss( $T_i$ ,  $x$ )
29     restore( $T_i$ ,  $x$ )
30     delete  $st_i(x)$ 
31      $lt_v(x) \leftarrow pv_i(x)$ 
32   }
33 }

34 procedure commit(Transaction  $T_i$ ) {
35   for ( $x : ASet(T_i)$  in parallel) {
36     wait until  $pv_i(x) - 1 = lt_v(x)$ 
37     dismiss( $T_i$ ,  $x$ )
38   }
39   if ( $\exists x$  in  $ASet(T_i)$  such that  $rv_i(x) > cv(x)$ )
40     abort( $T_i$ ) and exit
41   for ( $x : ASet(T_i)$  in parallel) {
42     delete  $st_i(x)$ 
43      $lt_v(x) \leftarrow pv_i(x)$ 
44   }
45 }

47 procedure checkpoint(Transaction  $T_i$ , Var  $x$ ) {
48   if ( $ac_i(x) = 0$ ) {
49     copy  $x$  to  $st_i(x)$ 
50      $rv_i(x) \leftarrow cv(x)$ 
51   }
52 }

54 procedure dismiss(Transaction  $T_i$ , Var  $x$ ) {
55   if ( $ac_i(x) = 0$  and  $rv_i(x) = cv(x)$ )
56      $cv(x) \leftarrow pv_i(x)$ 
57   if ( $pv_i(x) - 1 = lv(x)$ )
58      $lv(x) \leftarrow pv_i(x)$ 
59 }

61 procedure restore(Transaction  $T_i$ , Var  $x$ ) {
62   if ( $ac_i(x) \neq 0$  and  $rv_i(x) < cv(x)$ ) {
63     revert  $x$  from  $st_i(x)$ 
64      $cv(x) \leftarrow rv_i(x)$ 
65   }
66 }

```

Figure 23: SVA pseudocode.

extend the definition of closing write and closing read operation executions to a closing access execution which is such a read or write operation, that is not followed by any other closing read or closing write operation.

## 5.1 Counters

The basic premise of versioning algorithms is that counters are associated with transactions and used to allow or deny access by these transactions to shared objects (rather than only for recovery). SVA uses several version counters. The *private version* counter  $pv_i(x)$  uniquely defines the version of transaction  $T_i$  with respect to variable  $x$ . The *global version* counter  $gv(x)$  shows how many transactions that have  $x$  in their access set have started. The *local version* counter  $lv(x)$  shows which transaction can currently and exclusively access variable  $x$ . Specifically, the transaction that can access  $x$  is such  $T_i$  whose  $pv_i(x)$  is one greater than

$lv(x)$  (we refer to this as the *access condition*). The *local terminal version* counter  $ltv(x)$  shows which of the transactions that have  $x$  in their access set can currently commit or abort. That is,  $T_i$  such that  $x \in ASet(T_i)$  can commit or abort if its  $pv_i(x)$  is one greater than  $ltv(x)$ .

In addition, the *current version* counter  $cv(x)$  defines what state the variable is in and transactions that operate on  $x$  will increment its  $cv(x)$  to indicate a change of state. It will also revert the counter to indicate a rollback (abort). A *recovery version* counter  $rv_i(x)$  indicates what state variable  $x$  was in prior to transaction  $T_i$ 's modifications. These counters are used together to detect whether a variable accessed by the current transaction was rolled back by some other transaction, requiring that the current transaction roll back as well. These counters also determine which transaction is responsible for reverting the state of the variable and to what state it should be reverted. In order to be able to revert the state of variable SVA also uses a *variable store*  $st$ , where transactions store copies of variables before modifying them.

In order to detect the closing use of a variable, SVA requires that suprema on accesses be given for each variable used by a transaction. These are given for each variable  $x$  in transaction  $T_i$ 's access set as  $supr_i(x)$ . If the supremum is unknown  $supr_i(x) = \infty$ . We assume that if a supremum on accessing some variable  $x$  is zero, then  $x$  is excluded from the transaction's access set. Then, *access counter*  $ac_i(obj)$  is used to track the actual number of accesses by  $T_i$  on  $x$  and to check when the supremum is reached, to release a variable early.

Finally, SVA uses a map of locks  $lk$ , containing one lock for each variable to make a globally consistent snapshots. Locks are acquired and released in order  $<_{lk}$  to prevent deadlocks. Initially, all locks are unlocked, counters are set to zero, and the variable store is empty.

## 5.2 Transactions

The pseudocode for SVA is shown in Fig. 23. The life cycle of every SVA transaction begins with procedure **start** (we also refer to this part as initialization). Following that, a transaction may execute one or more accesses (reads or writes) to shared variables (procedure **access**). After any **access** or right after **start** a transaction may then either proceed to **commit** or **abort**, both of which end a transaction's life cycle. SVA transactions are prevented from committing until all preceding transactions which released their variables early and with which the current transaction shares variables also commit. Accesses to shared variables can be interleaved with various transaction-local operations, including accesses to non-shared structures or variables. However, those are only visible to the transaction to which they are local, so they do not influence other transactions. For the purpose of clarity, but without loss of generality, we omit those operations here. We also assume that transactions are executed in a single, fresh, dedicated thread. We also omit the concepts of nested and recurrent transactions.

**Start** The initialization of a transaction is shown in **start** at line 1. When transaction  $T_i$  starts it uses  $gv(x)$  to assign itself a unique version  $pv_i(x)$  for each variable  $x$  in its access set  $ASet(T_i)$ . This must be done atomically and in isolation, so these operations are guarded by locks—one lock  $lk(x)$  for each variable used.

**Accesses** Variables are accessed via procedure `access` at line 12. Before accessing a variable, transaction  $T_i$  waits for the access condition to be satisfied at line 13 (e.g. for the preceding transaction to release it). When this happens,  $T_i$  makes a backup copy using `checkpoint` (line 47). This procedure checks whether this is the first access, and if so makes a backup copy of  $x$  to  $\text{st}_i(x)$  at line 49 and sets the recovery counter to the current version of  $x$  at line 50 to indicate which version of  $x$  the transaction first viewed and can revert to in case of rollback (abort). Then, the access (either a write or read) is actually performed.

Afterward, transaction  $T_i$  increments the access counter  $\text{ac}_i(x)$  (line 18) and proceeds to check whether this was the closing access on  $x$  by comparing  $\text{ac}_i(x)$  to the appropriate supremum  $\text{supr}_i(x)$  (line 19). If this is the case, the variable is released early—i.e.,  $\text{lv}(x)$  is set to the same value as the transaction’s private counter  $\text{pv}_i(x)$  (line 21). At this point, some other transaction  $T_j$ , whose  $\text{pv}_j(x) - 1 = \text{lv}(x)$  can start accessing the variable using procedure `access`. Another provision made in SVA is that when variable  $x$  is released by transaction  $T_i$ ,  $\text{cv}(x)$  is set to the transaction’s  $\text{pv}_i(x)$  version (during early release at line 20). This signifies both that there is a new consistent version of  $x$  and that  $T_i$  modified  $x$  the most recently.

**Commit** Transaction  $T_i$  can attempt to commit using procedure `commit` at line 34. The variables in the transaction’s access set are committed independently (possibly in parallel). First, transaction  $T_i$  must pass the commit condition at line 36 for each variable in its access set, so that a transaction is not allowed to commit before all the transactions that accessed the same variables as  $T_i$  before  $T_i$  commit or abort. The transaction then checks whether it has access to the variable and if this is not the case, i.e., the variable was not accessed at all, waits until the preceding transaction releases it at line 43. Then, the committing transaction executes procedure `dismiss` at line 54. At this point, if the transaction did not access the variable before commit, the current version counter is updated (line 56). Furthermore, if this transaction has not previously released some variable  $x$ ,  $\text{lv}(x)$  is set to  $\text{pv}_i(x)$  to indicate the object is released (line 58). Finally, the transaction erases backup copies from the store  $\text{st}$  (line 42) and sets  $\text{ltv}(x)$  to its private version counter’s value (line 43) to indicate that a subsequent transaction can now perform `commit` or `abort` on  $x$ .

**Abort** Aborts are not performed by SVA as part of its basic *modus operandi*, but aborts can be triggered manually by the programmer. Furthermore, such manually-triggered aborts can also cause other transactions to abort. If the programmer decides to abort a transaction, or if an abort is forced, procedure `abort` (line 25) is used. As with commit, in order for the abort to proceed, the transaction must pass the commit condition at line 27 for each variable in its access set. Then, previously unreleased variables are released using `dismiss` (as described above). Afterward, the transaction restores its variables using procedure `restore` (line 61). There, if  $T_i$  accessed  $x$  at least once and the version of  $x$  that it stored prior to accessing  $x$  ( $\text{rv}_i(x)$ ) is lower than the current version of  $x$  ( $\text{cv}(x)$ ), then  $T_i$  is responsible for reverting  $x$  to the previous state (condition at line 62). In that case, this procedure reverts  $x$  to a copy from  $\text{st}_i(x)$  (line 63). It also sets the current version to  $\text{rv}_i(x)$  (line 64), indicating that variable  $x$  was restored to the state just from before  $T_i$  modified it. Note that this means that  $\text{cv}(x)$  was set to a lower value than before. Finally, the transaction cleans up  $\text{st}_i(x)$  (line 30) and sets  $\text{ltv}(x)$  to its private version counter’s value (line 31). As with `commit`, this procedure may operate on each variable in parallel.

If a manual abort is triggered by the user it may be the case that transaction  $T_i$  releases variable  $x$  and aborts after some other transaction  $T_j$  already reads the value written to  $x$

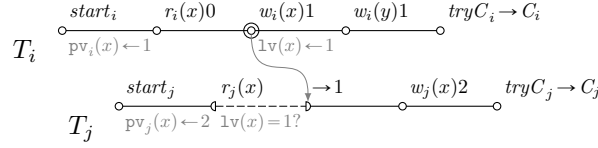


Figure 24: SVA early release example. The symbol  $\circ \dashrightarrow$  denotes an operation execution split into the invocation event and the response event to indicate waiting and  $\curvearrowright$  indicates what caused the wait.

by  $T_i$ . In that case  $T_j$  must be forced to abort along with  $T_i$ . Hence, if subsequently the situation arises that some other transaction  $T_j$  tries to access  $x$  after  $T_i$  released it early and aborted, there is a condition at line 15 in procedure **access** which will compare the values of  $\text{cv}(x)$  and  $\text{rv}_i(x)$  for  $x$ . If  $\text{cv}(x)$  and  $\text{rv}_j(x)$  are equal, that implies  $T_j$  currently has access to a consistent state of  $x$  and can access it. However, if  $\text{rv}_j(x)$  is greater than  $\text{cv}(x)$ , this means that some previous transaction (i.e.,  $T_i$ ) set  $\text{cv}(x)$  to a lower value than it was before, which means it aborted and reverted  $x$ . This causes  $T_j$  to be forced to abort (line 16) instead of accessing  $x$ . As an aside, if  $\text{cv}(x)$  is greater than  $\text{rv}_j(x)$ , then  $T_j$  already stopped modifying  $x$  and released it, so there should not be any more accesses to  $x$ . This means that  $T_j$  violated its upper bound for  $x$  and must also be forced to abort.

Similarly, if transaction  $T_j$  tries to commit after  $T_i$  aborted, then it checks the condition at line 39 for each variable, and if there is at least one variable  $x$  for which  $\text{rv}_j(x)$  is greater than  $\text{cv}(x)$ , then again some previous  $T_i$  must have aborted and reverted  $x$  and  $T_j$  must then abort too. Note that aborted transactions revert variables in the order imposed by the commit condition in wait (line 27), which ensures state consistency after rollback.

### 5.3 Examples

To illustrate further, the examples in Fig. 24–27 show some scenarios of interacting SVA transactions. In Fig. 24, transactions  $T_i$  and  $T_j$  both try to increment variable  $x$ . Since  $T_i$  starts before  $T_j$ , it has a lower version for  $x$  (e.g.,  $\text{pv}_i(x) \leftarrow 1$ ) than  $T_j$  (e.g.,  $\text{pv}_j(x) \leftarrow 2$ ). So, when accessing  $x$ ,  $T_i$  will pass its access condition for  $x$  sooner than  $T_j$ . Thus,  $T_j$  has to wait, while  $T_i$  executes its operations on  $x$ . After its last operation on  $x$ ,  $T_i$  releases it by setting  $x$ 's local counter to its own version ( $\text{lv}(x) \leftarrow 1$ ). From this moment  $T_i$  can no longer access  $x$ . But since the local counter is now equal to 1, which is one lower than  $T_j$ 's version for  $x$  of 2,  $T_j$  can now pass the access condition for  $x$  and start executing operations. In effect  $T_i$  and  $T_j$  can execute in parallel in part. Fig. 25 shows a similar example of a transaction  $T_i$  releasing  $x$  early and operating on  $y$  while a later transaction  $T_j$  operates on  $x$  in parallel. The scenario differs from the last in that  $T_j$  finishes work before  $T_i$ , however it still waits with committing until  $T_i$  commits. Fig. 26 shows another similar scenario, except  $T_i$  eventually aborts. Then,  $T_j$  is also forced to abort (and retry) by SVA. Note that if meanwhile  $T_j$  released  $x$  early and some transaction  $T_k$  accessed it,  $T_k$  would also be aborted. Thus, a cascading abort including multiple transactions is a possibility. Finally, Fig. 27 shows two transactions operating on different variables. Since SVA performs synchronization per variable, if the access sets of two transactions do not intersect, they can both execute in parallel, without any synchronization.

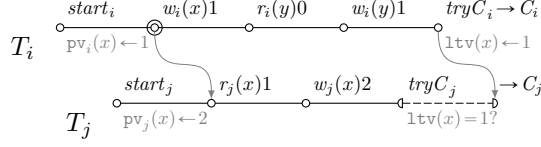


Figure 25: SVA wait on commit example.

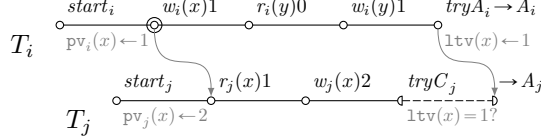


Figure 26: SVA cascading abort example.

## 5.4 Safety of SVA

We present a proof sketch showing that SVA is last-use opaque. A complete proof is in Appendix D. First, we make the following straightforward observations about SVA.

**Observation 1** (Version Order). *Given the set  $\mathbb{T}_H^x$  of all transactions that access  $x$  in  $H$  there is a total order called a version order  $<_x$  on  $\mathbb{T}_H^x$  s.t. for any  $T_i, T_j \in \mathbb{T}_H^x$ ,  $T_i <_x T_j$  if  $\text{pv}_i(x) < \text{pv}_j(x)$ .*

**Observation 2** (Access Order). *If  $T_i <_x T_j$  and  $T_i$  performs operation  $op_i$  on  $x$ , and  $T_j$  performs operation  $op_j$  on  $x$ , then  $op_i$  is completed in  $H$  before  $op_j$ .*

**Observation 3** (No Bufferring). *Since transactions operate on variables rather than buffers, any read operation  $op = r_i(x) \rightarrow v$  in any transaction  $T_i$  is preceded in  $H$  by some write operation  $w_j(x)v \rightarrow ok_j$  in some  $T_j$  (possibly  $i = j$ ).*

**Observation 4** (Read from Released). *If transaction  $T_i$  executes a read operation or a write operation  $op$  on  $x$  in  $H$ , then any transaction that previously executed a read or write operation on  $x$  is either committed, aborted, or decided on  $x$  before  $op$ .*

**Observation 5** (Do not Read Aborted). *Assuming unique writes, if transaction  $T_i$  executes  $w_i(x)v \rightarrow u$  and aborts in  $H$ , then  $x$  will be reverted to a previous value. In consequence, no other transaction can read  $v$  from  $x$ .*

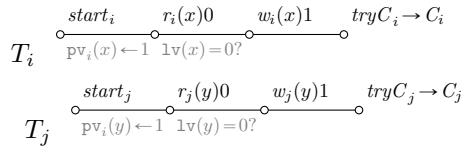


Figure 27: SVA parallel execution example.

**Observation 6** (Commit Order). *If transaction  $T_i$  accesses  $x$  in  $H$  and commits or aborts in  $H$ , any transaction that previously executed a read or write operation on  $x$  is either committed or aborted before  $T_i$  commits or aborts.*

**Observation 7** (Forced Abort). *If transaction  $T_i$  reads  $x$  from  $T_j$  and  $T_j$  subsequently aborts, then  $T_i$  also aborts.*

Then, the main lemma follows, showing that SVA produces final-state opaque histories. For convenience, we assume that the SVA program always writes values to variables that are unique and in the domain of the variable.

**Lemma 11.** *Any SVA history  $H$  is final-state last-use opaque.*

*sketch.* Let  $H_C = \text{Compl}(H)$  be a completion of  $H$  if for every  $T_i \in H$ , if  $T_i$  is live or commit-pending in  $H$ , then  $T_i$  is aborted in  $H_C$ . Given  $H_C$  we can construct  $\hat{S}_H$ , a sequential history s.t.  $\hat{S}_H \equiv H_C$ , where for any two transactions  $T_i, T_j \in H_C$ :

- a) if  $T_i <_{H_C} T_j$ , then  $T_i <_{\hat{S}_H} T_j$ ,
- b) if  $T_i <_x T_j$  for any variable  $x$ , then  $T_i <_{\hat{S}_H} T_j$ .

Note that if some transaction  $T_i$  commits in  $H$ , then it commits in  $\hat{S}_H$  (and *vice versa*). Otherwise  $T_i$  aborts in  $\hat{S}_H$ .

Let  $T_i$  be any transaction committed in  $H$ . Thus,  $T_i$  also commits in  $\hat{S}_H$ . From Observation 3, any read operation execution  $op_i = r_i(x) \rightarrow v$  in  $H|T_i$  is preceded in  $H$  by  $op_j = w_j(x)v \rightarrow ok_j$ . If  $op_i$  is local, then  $i = j$ , so  $op_j$  is in a committed transaction. If  $op_i$  is not local, then  $i \neq j$ . In that case, from Observation 5,  $T_j$  cannot be aborted before  $op_i$  in  $H$ . Consequently,  $T_j$  is either committed before  $op_i$  in  $H$ , live in  $H$ , or committed or aborted after  $op_i$ . In the former case  $T_i$  reads from a committed transaction. In the latter case, since  $T_i$  is committed, then from Observation 4 and Observation 6 we know that  $T_j$  commits or aborts in  $H$  before  $T_i$  commits. In addition, from Observation 7 we know that  $T_j$  cannot abort in  $H$ , because it would have caused  $T_i$  to also abort. Thus, any committed  $T_i$  reads only from committed transactions.

From Observation 2, if  $T_i$  reads from the value written by an operation in  $T_j$  then the write in  $T_j$  completes before the read in  $T_i$ , which implies  $T_j <_x T_i$ . Hence,  $T_j <_{\hat{S}_H} T_i$ . Thus, if  $T_i$  is committed in  $\hat{S}_H$  and reads from some  $T_j$ , then any such  $T_j$  is committed and precedes  $T_i$ , so  $\hat{S}_H|T_j \subseteq \text{Vis}(\hat{S}_H, T_i)$ . Since all reads in committed transactions read from preceding committed transactions, then for each read in  $\text{Vis}(\hat{S}_H, T_i)$  reading  $v$  from  $x$  there will be a write operation execution writing  $v$  to  $x$  in  $\text{Vis}(\hat{S}_H, T_i)$ . Since, from Observation 2 all accesses on  $x$  operations follow  $<_x$ , then  $\text{Vis}(\hat{S}_H, T_i)$  is legal for any committed  $T_i$ . Thus, any  $T_i$  that is committed in  $\hat{S}_H$  is legal in  $\hat{S}_H$ .

Let  $T_i$  be a transaction that is live or aborts in  $H$ , so it aborts in  $\hat{S}_H$ . From Observation 3 any read operation execution  $op_i = r_i(x) \rightarrow v$  in  $H|T_i$  is preceded in  $H$  by  $op_j = w_j(x)v \rightarrow ok_j$ . If  $op_i$  is local, then  $i = j$ , so  $op_j$  is always in  $\text{Vis}(\hat{S}_H, T_i)$  where  $op_j$  precedes  $op_i$ . If  $op_i$  is not local, then  $i \neq j$ . In that case, from Observation 5,  $T_j$  cannot be aborted before  $op_i$  in  $H$ . Consequently,  $T_j$  is either committed before  $op_i$  in  $H$ , live in  $H$ , or committed or aborted after  $op_i$ . In the former case  $T_i$  reads from a committed transaction. In the latter case, from Observation 4 we know that either  $T_j$  commits in  $H$  or  $T_j$  is decided on  $x$  in  $H$ . Thus, any committed  $T_i$  reads  $x$  only from committed transactions or transactions that are decided on  $x$ .

From Observation 2, if  $T_i$  reads from the value written by an operation in  $T_j$  then the write in  $T_j$  completes before the read in  $T_i$ , which implies  $T_j <_x T_i$ . Hence,  $T_j <_{\hat{S}_H} T_i$ . Thus, if  $T_i$  is aborted in  $\hat{S}_H$  and reads from some  $T_j$ , then any such  $T_j$  is either committed and precedes  $T_i$ , so  $\hat{S}_H|T_j \subseteq LVis(\hat{S}_H, T_i)$ , or  $T_j$  is decided on any  $x$  if  $T_i$  reads from  $x$ , so  $\hat{S}_H|T_j \subseteq LVis(\hat{S}_H, T_i)$ . Since all reads in aborted transactions read  $x$  from preceding committed transactions or transactions decided on  $x$ , then for each read in  $LVis(\hat{S}_H, T_i)$  reading  $v$  from  $x$  there will be a write operation execution writing  $v$  to  $x$ . Since, from Observation 2 all accesses on  $x$  operations follow  $<_x$ , then  $LVis(\hat{S}_H, T_i)$  is legal for any aborted  $T_i$ . Thus, any  $T_i$  that is aborted in  $\hat{S}_H$  is last-use legal in  $\hat{S}_H$ .

Since any committed  $T_i$  in  $\hat{S}_H$  is legal in  $\hat{S}_H$ , and any aborted  $T_i$  in  $\hat{S}_H$  is last-use legal in  $\hat{S}_H$ , and since  $\hat{S}_H$  trivially follows the real time order of  $H$ , then from Def. 21  $H$  is final-state last-use opaque.  $\square$

**Theorem 16.** *Any SVA history  $H$  is last-use opaque.*

*Proof.* Since by Lemma 11 any SVA history  $H$  is final-state last-use opaque, and any prefix  $P$  of  $H$  is also an SVA history, then every prefix of  $H$  is also final-state last-use opaque. Thus, by Def. 22,  $H$  is last-use opaque.  $\square$

## 6 Related Work

Ever since opacity [14, 15] was introduced, it seems, there were attempts to weaken its stringent requirements, while retaining some guarantees over what serializability [6, 25] provides. We explore the most pertinent examples in Section 3: TMS1, TMS2 [11], elastic opacity [13], live opacity [12], and VWC [21], as well as some apposite consistency criteria: recoverability [16], ACA [7], strictness [7], and rigorousness [9]. Other attempts were more specialized and include virtual time opacity [21], where the real-time order condition is relaxed. Similarly, the  $\diamond$  opacity family of properties [22] relax the time ordering requirements of opacity to provide properties applicable to deferred update replication. Another example is view transactions [1], where it is only required that a transaction commits on any snapshot, that can be different than the one the transaction viewed initially, provided that operating on either snapshot produced externally indistinguishable results. While these properties have specific applications, none weaken the consistency to allow variable access before commit.

Although algorithms and systems are not the focus of this paper, some systems research that explores relaxed consistency should be noted. We already mention our own SVA [31] in Section 5. Dynamic STM [19] is another system with early release, and it can be credited with introducing the concept of early release in the TM context. Dynamic STM allows transactions that only perform read operations on particular variables to (manually) release them for use by other transactions. However, it left the assurance of safety to the programmer, and, as the authors state, even linearizability cannot be guaranteed by the system. The authors of [35] expanded on the work above and evaluated the concept of early release with respect to read-only variables on several concurrent data structures. The results showed that this form of early release does not provide a significant advantage in most cases, although there are scenarios where it would be advantageous if it were automated. We use a different approach in SVA, where early release is not limited to read-only variables. Twilight STM [8] relaxes isolation to allow transactions to read variables that are used by other transactions, and allow them to re-read their values as they change in order to maintain

the correctness of the computation. If inconsistencies arise, a reconciliation is attempted on commit, and aborts are induced if this is impossible. Since it allows operating on variables that were released early, but potentially before closing write, Twilight STM will not satisfy the consistency requirements of last-use opacity, but it is likely to guarantee serializability and recoverability.

DATM [26] is yet another noteworthy system with an early release mechanism. DATM is an optimistic multicore-oriented TM based on TL2 [10], augmented with early-release support. It allows a transaction  $T_i$  to write to a variable that was accessed by some uncommitted transaction  $T_j$ , as long as  $T_j$  commits before  $T_i$ . DATM also allows transaction  $T_i$  to read a speculative value, one written by  $T_j$  and accessed by  $T_i$  before  $T_j$  commits. DATM detects if  $T_j$  overwrites the data or aborts, in which case  $T_i$  is forced to restart. DATM allows all schedules allowed by conflict-serializability. This means that DATM allows overwriting, as well as cascading aborts. It also means that it does not satisfy last-use opacity.

## 7 Conclusions

This paper explored the space of TM safety properties in terms of whether or not, and to what extent, they allow a transaction to release a variable early, or, in other words, for a transaction to read a value written by a live transaction. We showed that existing properties are either strong, but prevent early release altogether (opacity, TMS1 and TMS2), or pose impractical restrictions on the ability of transactions to abort (VWC and live opacity). The remainder of the properties are not strong enough for TM applications (serializability and recoverability) since they allow a large range of inconsistent views, including overwriting. Hence, we presented a new TM safety property called last-use opacity that strikes a reasonable compromise. It allows early release without a requirement for transactions that release early not to abort, but one that is nevertheless strong enough to prevent most inconsistent views and make others inconsequential. The resulting property may be a useful practical criterion for reasoning about TMs with early release support.

We discussed the histories that are allowed by last-use opacity and examined the guarantees the property gives to the programmer. Last-use opacity always allows for potential inconsistent views to occur due to cascading aborts. However, no other inconsistent views are allowed. The inconsistent views that can occur can be made harmless by taking away the programmer’s ability to execute arbitrary aborts by either removing that operation completely or by removing it from the programmer’s toolkit, but allowing it to be used by the TM system, e.g. for fault tolerance. Allowing the programmer to abort a transaction at will means that they will need to eliminate dangerous situations (possible division by zero, invalid memory accesses, etc.) on a case-by-case basis. Nevertheless, we predict that inconsistent views of this sort will be relatively rare in practical TM applications, and typically result from using the abort operation to program business logic. Alternatively, a variant of last-use opacity called  $\beta$ -last-use opacity can be used instead, which eliminates the inconsistent views by preventing early release in transactions where a programmatic abort is possible.

Finally, we discussed SVA, a pessimistic concurrency control TM algorithm with early release, which we show satisfies last-use opacity.

**Acknowledgements** The project was funded from National Science Centre funds granted by decision No. DEC-2012/06/M/ST6/00463.

## References

- [1] Y. Afek, A. Morrison, and M. Tzafrir. Brief Announcement: View Transactions: Transactional Model with Relaxed Consistency Checks. In *Proceedings of PODC'10: the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, July 2010.
- [2] H. Attiya, A. Gotsman, S. Hans, and N. Rinetzky. A programming language perspective on transactional memory consistency. In *Proceedings of PODC'13: the 32nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 2013.
- [3] H. Attiya, A. Gotsman, S. Hans, and N. Rinetzky. Safety of live transactions in transactional memory: Tms is necessary and sufficient. In *Proceedings of DISC'14: the 28th International Symposium on Distributed Computing*, 2014.
- [4] H. Attiya and S. Hans. Transactions are Back—but How Different They Are? In *Proceedings of TRANSACT'14: the 7th ACM SIGPLAN Workshop on Transactional Computing*, Feb. 2014.
- [5] H. Attiya, S. Hans, P. Kuznetsov, and S. Ravi. Safety of Deferred Update in Transactional Memory. In *Proceedings of ICDCS'13: the 33rd International Conference on Distributed Computing Systems*, July 2013.
- [6] P. Bernstein, D. Shipman, and W. Wong. Formal Aspects of Serializability in Database Concurrency Control. *IEEE Transactions on Software Engineering*, SE-5(3):203–216, May 1979.
- [7] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, 1987.
- [8] A. Bieniusa, A. Middelkoop, and P. Thiemann. Brief Announcement: Actions in the Twilight—Concurrent Irrevocable Transactions and Inconsistency Repair. In *Proceedings of PODC'10: the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, July 2010.
- [9] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz. On rigorous transaction scheduling. *IEEE Transactions on Software Engineering*, 17, Sept. 1991.
- [10] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of DISC'06: the 20th International Symposium on Distributed Computing*, Sept. 2006.
- [11] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, 25:769–799, Sept. 2013.
- [12] D. Dziuwa, P. Fatourou, and E. Kanellou. Consistency for transactional memory computing. *Bulletin of the EATCS*, 113, 2014.

- [13] P. Felber, V. Gramoli, and R. Guerraoui. Elastic Transactions. In *Proceedings of DISC'09: the 23rd International Symposium on Distributed Computing*, Sept. 2009.
- [14] R. Guerraoui and M. Kapalka. On the Correctness of Transactional Memory. In *Proceedings of PPOPP'08: the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 175–184, Feb. 2008.
- [15] R. Guerraoui and M. Kapalka. *Principles of Transactional Memory*. Morgan & Claypool, 2010.
- [16] V. Hadzilacos. A theory of reliability in database systems. *Journal of the ACM*, 35, Jan. 1988.
- [17] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Proceedings of OOPSLA'03: the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2003.
- [18] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of PPOPP'05: the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2005.
- [19] M. Herlihy, V. Luchangco, M. Moir, and I. W. N. Scherer. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of PODC'03: the 22nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 92–101, July 2003.
- [20] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of ISCA'93: the 20th International Symposium on Computer Architecture*, pages 289–300, May 1993.
- [21] D. Imbs, J. R. de Mendivil, and M. Raynal. On the Consistency Conditions of Transactional Memories. Technical Report 1917, IRISA, Dec. 2008.
- [22] T. Kobus, M. Kokociński, and P. T. Wojciechowski. The correctness criterion for deferred update replication. In *Proceedings of TRANSACT '15: the 10th ACM SIGPLAN Workshop on Transactional Computing*, 2015.
- [23] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2), Mar. 1977.
- [24] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for C/C++. In *Proceedings of OOPSLA'08: the 23rd ACM SIGPLAN Conference on Object-oriented Programming, Systems Languages and Applications*, 2008.
- [25] C. H. Papadimitrou. The Serializability of Concurrent Database Updates. *Journal of the ACM*, 26(4):631–653, 1979.
- [26] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing Conflicting Transactions in an STM. In *Proceedings of PPOPP'09: the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2009.

- [27] M. F. Ringenburt and D. Grossman. AtomCaml: first-class atomicity via rollback. In *Proceedings of ICFP'05: the 10th ACM SIGPLAN International Conference on Functional Programming*, Sept. 2005.
- [28] N. Shavit and D. Touitou. Software Transactional Memory. In *Proceedings of PODC'95: the 14th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 204–213, Aug. 1995.
- [29] K. Siek and P. T. Wojciechowski. A Formal Design of a Tool for Static Analysis of Upper Bounds on Object Calls in Java. In *Proceedings of FMICS'12: the 17th International Workshop on Formal Methods for Industrial Critical Systems*, number 7437 in Lecture Notes in Computer Science, pages 192–206, Aug. 2012.
- [30] K. Siek and P. T. Wojciechowski. Brief announcement: Towards a Fully-Articulated Pessimistic Distributed Transactional Memory. In *Proceedings of SPAA'13: the 25th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 111–114, July 2013.
- [31] K. Siek and P. T. Wojciechowski. Atomic RMI: a Distributed Transactional Memory Framework. In *Proceedings of HLPP'14: the 7th International Symposium on High-level Parallel Programming and Applications*, pages 73–94, July 2014.
- [32] K. Siek and P. T. Wojciechowski. Brief announcement: Relaxing opacity in pessimistic transactional memory. In *Proceedings of DISC'14: the 28th International Symposium on Distributed Computing*, 2014.
- [33] K. Siek and P. T. Wojciechowski. Zen and the Art of Concurrency Control: An Exploration of TM Safety Property Space with Early Release in Mind. In *Proceedings of WTTM'14: the 6th Workshop on the Theory of Transactional Memory*, July 2014.
- [34] K. Siek and P. T. Wojciechowski. Atomic RMI: A Distributed Transactional Memory Framework. *International Journal of Parallel Programming*, 2015.
- [35] T. Skare and C. Kozyrakis. Early release: Friend or foe? In *Proceedings of WTW'06: the Workshop on Transactional Memory Workloads*, June 2006.
- [36] W. E. Weihl. Local atomicity properties: modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems*, 11(2):249–282, Apr. 1989.
- [37] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable Transactions and their Applications. In *Proceedings of SPAA'08: the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, June 2008.
- [38] P. T. Wojciechowski. Isolation-only Transactions by Typing and Versioning. In *Proceedings of PPDP'05: the 7th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming*, July 2005.
- [39] P. T. Wojciechowski. *Language Design for Atomicity, Declarative Synchronization, and Dynamic Update in Communicating Systems*. Publishing House of Poznań University of Technology, 2007.

- [40] P. T. Wojciechowski, O. Rütli, and A. Schiper. SAMOA: A framework for a synchronisation-augmented microprotocol approach. In *Proceedings of IPDPS '04: the 18th IEEE International Parallel and Distributed Processing Symposium*, Apr. 2004.

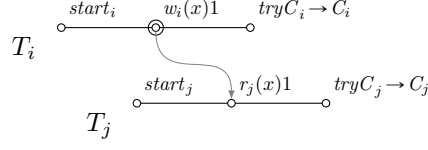


Figure 28: History  $H_1$ , last-use opaque.

## A Last-use Opacity History Examples

By  $A \sqsubseteq B$  we denote that sequence  $A$  is a substring of  $B$ .

Note the following about  $Seq(x)$  for any  $x, T_i, T_j$ :

$$[r_i(x) \rightarrow 0] \in Seq(x) \quad (1)$$

$$[w_i(x)1 \rightarrow ok_i] \in Seq(x) \quad (2)$$

$$[w_i(x)1 \rightarrow A_i] \in Seq(x) \quad (3)$$

$$[w_i(x)1 \rightarrow ok_i, r_j(x) \rightarrow 1] \in Seq(x) \quad (4)$$

$$[w_i(x)1 \rightarrow ok_i, r_j(x) \rightarrow A_i] \in Seq(x) \quad (5)$$

$$[w_i(x)1 \rightarrow ok_i, r_j(x) \rightarrow 1, w_j(x)2 \rightarrow ok_j] \in Seq(x) \quad (6)$$

$$[w_i(x)1 \rightarrow ok_i, r_j(x) \rightarrow 1, w_j(x)2 \rightarrow A_j] \in Seq(x) \quad (7)$$

$$\emptyset \in Seq(x) \quad (8)$$

$$[r_i(x) \rightarrow 1] \notin Seq(x) \quad (9)$$

$$[w_i(x)1 \rightarrow ok_i, w_i(x)2 \rightarrow ok_i, r_i(x) \rightarrow 1] \notin Seq(x) \quad (10)$$

$$[w_i(x)1 \rightarrow ok_i, r_i(y) \rightarrow 1, r_j(x) \rightarrow 1, w_j(y)1 \rightarrow ok_j] \notin Seq(x) \quad (11)$$

**Lemma 12.**  $H_1$  is final-state last-use opaque.

*Proof.*

$$\text{let } C_1 = \text{Compl}(H_1) = H_1 \quad (12)$$

$$\text{let } S_1 = C_1|T_i \cdot C_1|T_j \quad (13)$$

$$S_1 \equiv C_1 \quad (14)$$

$$\text{real time order } <_{H_1} = \emptyset \quad (15)$$

$$\text{real time order } <_{S_1} = \{T_i <_{S_1} T_j\} \quad (16)$$

$$(15) \wedge (16) \implies <_{S_1} \subseteq <_{H_1} \quad (17)$$

$$i = i \implies S_1|T_i \subseteq \text{Vis}(S_1, T_i) \quad (18)$$

$$T_i <_{S_1} T_j \implies S_1|T_j \not\subseteq \text{Vis}(S_1, T_i) \quad (19)$$

$$(18) \wedge (19) \implies \text{Vis}(S_1, T_i) = S_1|T_i \quad (20)$$

$$(20) \implies \text{Vis}(S_1, T_i)|x = [w_i(x)1 \rightarrow ok_i] \quad (21)$$

$$(21) \wedge (2) \implies \text{Vis}(S_1, T_i) \text{ is legal} \quad (22)$$

$$(22) \implies T_i \text{ in } S_1 \text{ is legal in } S_1 \quad (23)$$

$$T_i <_{S_1} T_j \wedge \text{res}_i(C_i) \in S_1|T_i \implies S_1|T_i \subseteq \text{Vis}(S_1, T_i) \quad (24)$$

$$j = j \implies S_1|T_j \subseteq \text{Vis}(S_1, T_i) \quad (25)$$

$$(24) \wedge (25) \implies \text{Vis}(S_1, T_j) = S_1|T_i \cdot S_1|T_j \quad (26)$$

$$(26) \implies \text{Vis}(S_1, T_j)|x = [w_i(x)1 \rightarrow ok_i, r_j(x) \rightarrow 1] \quad (27)$$

$$(27) \wedge (4) \implies \text{Vis}(S_1, T_j) \text{ is legal} \quad (28)$$

$$(28) \implies T_j \text{ in } S_1 \text{ is legal in } S_1 \quad (29)$$

$$(14) \wedge (17) \wedge (23) \wedge (29) \implies H_1 \text{ is final-state last-use opaque} \quad (30)$$

□

Let  $P_1^1$  be a prefix s.t.  $H_1 = P_1^1 \cdot [\text{res}_j(C_j)]$ .

**Lemma 13.**  $P_1^1$  is final-state last-use opaque.

*Proof.*

$$\text{let } C_1^1 = \text{Compl}(P_1^1) = P_1^1 \cdot [\text{res}_j(C_j)] \quad (31)$$

$$H_1 = C_1^1 \wedge \text{Lemma 12} \implies P_1^1 \text{ is final-state last-use opaque} \quad (32)$$

□

Let  $P_2^1$  be a prefix s.t.  $H_1 = P_2^1 \cdot [\text{try}C_j \rightarrow C_j]$ .

**Lemma 14.**  $P_2^1$  is final-state last-use opaque.

*Proof.*

$$\text{let } C_2^1 = \text{Compl}(P_2^1) = P_2^1 \cdot [\text{try}A_j \rightarrow A_j] \quad (33)$$

$$\text{let } S_2^1 = C_2^1|T_i \cdot C_2^1|T_j \quad (34)$$

$$S_2^1 \equiv C_2^1 \quad (35)$$

$$\text{real time order } <_{P_2^1} = \emptyset \quad (36)$$

$$\text{real time order } <_{S_2^1} = \{T_i <_{S_2^1} T_j\} \quad (37)$$

$$(36) \wedge (37) \implies <_{S_2^1} \subseteq <_{P_2^1} \quad (38)$$

$$i = i \implies S_2^1|T_i \subseteq \text{Vis}(S_2^1, T_i) \quad (39)$$

$$T_i <_{S_2^1} T_j \implies S_2^1|T_j \not\subseteq \text{Vis}(S_2^1, T_i) \quad (40)$$

$$(39) \wedge (40) \implies \text{Vis}(S_2^1, T_i) = S_2^1|T_i \quad (41)$$

$$(41) \implies \text{Vis}(S_2^1, T_i)|x = [w_i(x)1 \rightarrow ok_i] \quad (42)$$

$$(42) \wedge (2) \implies \text{Vis}(S_2^1, T_i) \text{ is legal} \quad (43)$$

$$(43) \implies T_i \text{ in } S_2^1 \text{ is legal in } S_2^1 \quad (44)$$

$$T_i <_{S_2^1} T_j \wedge \text{res}_i(C_i) \in S_2^1|T_i \implies S_2^1|T_i \subseteq \text{LVis}(S_2^1, T_j) \quad (45)$$

$$j = j \implies S_2^1|T_j \subseteq \text{LVis}(S_2^1, T_j) \quad (46)$$

$$(45) \wedge (46) \implies \text{LVis}(S_2^1, T_j) = S_2^1|T_i \cdot S_2^1|T_j \quad (47)$$

$$(47) \implies \text{LVis}(S_2^1, T_j)|x = [w_i(x)1 \rightarrow ok_i, r_j(x) \rightarrow 1] \quad (48)$$

$$(48) \wedge (4) \implies \text{LVis}(S_2^1, T_j) \text{ is legal} \quad (49)$$

$$(49) \implies T_j \text{ in } S_2^1 \text{ is last-use legal in } S_2^1 \quad (50)$$

$$(35) \wedge (38) \wedge (44) \wedge (50) \implies P_2^1 \text{ is final-state last-use opaque} \quad (51)$$

□

Let  $P_3^1$  be a prefix s.t.  $H_1 = P_3^1 \cdot [\text{res}_i(C_i), \text{try}C_j \rightarrow C_j]$ .

**Lemma 15.**  $P_3^1$  is final-state last-use opaque.

*Proof.*

$$\text{let } C_3^1 = \text{Compl}(P_3^1) = P_3^1 \cdot [\text{res}_i(C_i), \text{try}C_j \rightarrow C_j] \quad (52)$$

$$P_2^1 = C_3^1 \wedge \text{Lemma 14} \implies P_3^1 \text{ is final-state last-use opaque} \quad (53)$$

□

Let  $P_4^1$  be a prefix s.t.  $H_1 = P_4^1 \cdot [\text{try}C_i \rightarrow C_i, \text{try}C_j \rightarrow C_j]$ .

**Lemma 16.**  $P_4^1$  is final-state last-use opaque.

*Proof.*

$$\text{let } C_4^1 = \text{Compl}(P_4^1) = P_4^1 \cdot [\text{try}A_i \rightarrow A_i, \text{try}A_j \rightarrow A_j] \quad (54)$$

$$\text{let } S_4^1 = C_4^1|T_i \cdot C_4^1|T_j \quad (55)$$

$$S_4^1 \equiv C_4^1 \quad (56)$$

$$\text{real time order } <_{P_4^1} = \emptyset \quad (57)$$

$$\text{real time order } <_{S_4^1} = \{T_i <_{S_4^1} T_j\} \quad (58)$$

$$(57) \wedge (58) \implies <_{S_4^1} \subseteq <_{P_4^1} \quad (59)$$

$$i = i \implies S_4^1|T_i \subseteq \text{LVis}(S_4^1, T_i) \quad (60)$$

$$T_i <_{S_4^1} T_j \implies S_4^1|T_j \not\subseteq \text{LVis}(S_4^1, T_i) \quad (61)$$

$$(60) \wedge (61) \implies \text{LVis}(S_4^1, T_i) = S_4^1|T_i \quad (62)$$

$$(62) \implies \text{Vis}(S_4^1, T_i)|x = [w_i(x)1 \rightarrow ok_i] \quad (63)$$

$$(63) \wedge (2) \implies \text{LVis}(S_4^1, T_i) \text{ is legal} \quad (64)$$

$$(64) \implies T_i \text{ in } S_4^1 \text{ is last-use legal in } S_4^1 \quad (65)$$

$$w_i(x)1 \rightarrow ok_i \text{ is closing write on } x \text{ in } T_i \implies T_i \text{ is decided on } xS_4^1 \quad (66)$$

$$T_i <_{S_4^1} T_j \wedge (66) \implies S_4^1|T_i \subseteq \text{LVis}(S_4^1, T_j) \quad (67)$$

$$j = j \implies S_4^1|T_j \subseteq \text{LVis}(S_4^1, T_j) \quad (68)$$

$$(67) \wedge (68) \implies \text{LVis}(S_4^1, T_j) = S_4^1|T_i \cdot S_4^1|T_j \quad (69)$$

$$(69) \implies \text{LVis}(S_4^1, T_j)|x = [w_i(x)1 \rightarrow ok_i, r_j(x) \rightarrow 1] \quad (70)$$

$$(70) \wedge (4) \implies \text{LVis}(S_4^1, T_j) \text{ is legal} \quad (71)$$

$$(71) \implies T_j \text{ in } S_4^1 \text{ is last-use legal in } S_4^1 \quad (72)$$

$$(56) \wedge (59) \wedge (65) \wedge (72) \implies P_4^1 \text{ is final-state last-use opaque} \quad (73)$$

□

Let  $P_5^1$  be a prefix s.t.  $H_1 = P_5^1 \cdot [\text{res}_i(1), \text{try}C_i \rightarrow C_i, \text{try}C_j \rightarrow C_j]$ .

**Lemma 17.**  $P_5^1$  is final-state last-use opaque.

*Proof.*

$$\text{let } C_5^1 = \text{Compl}(P_5^1) = P_5^1 \cdot [\text{res}_i^{A_i}(\cdot) \text{try} A_j \rightarrow A_j] \quad (74)$$

$$\text{let } S_5^1 = C_5^1 | T_i \cdot C_5^1 | T_j \quad (75)$$

$$S_5^1 \equiv C_5^1 \quad (76)$$

$$\text{real time order } <_{P_5^1} = \emptyset \quad (77)$$

$$\text{real time order } <_{S_5^1} = \{T_i <_{S_5^1} T_j\} \quad (78)$$

$$(77) \wedge (78) \implies <_{S_5^1} \subseteq <_{P_5^1} \quad (79)$$

$$i = i \implies S_5^1 | T_i \subseteq \text{LVis}(S_5^1, T_i) \quad (80)$$

$$T_i <_{S_5^1} T_j \implies S_5^1 | T_j \not\subseteq \text{LVis}(S_5^1, T_i) \quad (81)$$

$$(80) \wedge (81) \implies \text{LVis}(S_5^1, T_i) = S_5^1 | T_i \quad (82)$$

$$(82) \implies \text{Vis}(S_5^1, T_i) | x = [w_i(x) 1 \rightarrow ok_i] \quad (83)$$

$$(83) \wedge (2) \implies \text{LVis}(S_5^1, T_i) \text{ is legal} \quad (84)$$

$$(84) \implies T_i \text{ in } S_5^1 \text{ is last-use legal in } S_5^1 \quad (85)$$

$$w_i(x) 1 \rightarrow ok_i \text{ is closing write on } x \text{ in } T_i \implies T_i \text{ is decided on } x \quad (86)$$

$$T_i <_{S_5^1} T_j \wedge (86) \implies S_5^1 | T_i \subseteq \text{LVis}(S_5^1, T_j) \quad (87)$$

$$j = j \implies S_5^1 | T_j \subseteq \text{LVis}(S_5^1, T_j) \quad (88)$$

$$(87) \wedge (88) \implies \text{LVis}(S_5^1, T_j) = S_5^1 | T_i \cdot S_5^1 | T_j \quad (89)$$

$$(89) \implies \text{LVis}(S_5^1, T_j) | x = [w_i(x) 1 \rightarrow ok_i, r_j(x) \rightarrow A_i] \quad (90)$$

$$(90) \wedge (5) \implies \text{LVis}(S_5^1, T_j) \text{ is legal} \quad (91)$$

$$(91) \implies T_j \text{ in } S_5^1 \text{ is last-use legal in } S_5^1 \quad (92)$$

$$(76) \wedge (79) \wedge (85) \wedge (92) \implies P_5^1 \text{ is final-state last-use opaque} \quad (93)$$

□

Let  $P_6^1$  be a prefix s.t.  $H_1 = P_6^1 \cdot [r_1(x) \rightarrow 1, \text{try} C_i \rightarrow C_i, \text{try} C_j \rightarrow C_j]$ .

**Lemma 18.**  $P_6^1$  is final-state last-use opaque.

*Proof.*

$$\text{let } C_6^1 = \text{Compl}(P_6^1) = P_6^1 \cdot [\text{try}A_i \rightarrow A_i, \text{try}A_j \rightarrow A_j] \quad (94)$$

$$\text{let } S_6^1 = C_6^1|T_i \cdot C_6^1|T_j \quad (95)$$

$$S_6^1 \equiv C_6^1 \quad (96)$$

$$\text{real time order } <_{P_6^1} = \emptyset \quad (97)$$

$$\text{real time order } <_{S_6^1} = \{T_i <_{S_6^1} T_j\} \quad (98)$$

$$(97) \wedge (98) \implies <_{S_6^1} \subseteq <_{P_6^1} \quad (99)$$

$$i = i \implies S_6^1|T_i \subseteq \text{LVis}(S_6^1, T_i) \quad (100)$$

$$T_i <_{S_6^1} T_j \implies S_6^1|T_j \not\subseteq \text{LVis}(S_6^1, T_i) \quad (101)$$

$$(100) \wedge (101) \implies \text{LVis}(S_6^1, T_i) = S_6^1|T_i \quad (102)$$

$$(102) \implies \text{Vis}(S_6^1, T_i)|x = [w_i(x)1 \rightarrow ok_i] \quad (103)$$

$$(103) \wedge (2) \implies \text{LVis}(S_6^1, T_i) \text{ is legal} \quad (104)$$

$$(104) \implies T_i \text{ in } S_6^1 \text{ is last-use legal in } S_6^1 \quad (105)$$

$$w_i(x)1 \rightarrow ok_i \text{ is closing write on } x \text{ in } T_i \implies T_i \text{ is decided on } x \quad (106)$$

$$T_i <_{S_6^1} T_j \wedge (106) \implies S_6^1|T_i \subseteq \text{LVis}(S_6^1, T_j) \quad (107)$$

$$j = j \implies S_6^1|T_j \subseteq \text{LVis}(S_6^1, T_j) \quad (108)$$

$$(107) \wedge (108) \implies \text{LVis}(S_6^1, T_j) = S_6^1|T_i \cdot S_6^1|T_j \quad (109)$$

$$(109) \implies \text{LVis}(S_6^1, T_j)|x = [w_i(x)1 \rightarrow ok_i] \quad (110)$$

$$(110) \wedge (2) \implies \text{LVis}(S_6^1, T_j) \text{ is legal} \quad (111)$$

$$(111) \implies T_j \text{ in } S_6^1 \text{ is last-use legal in } S_6^1 \quad (112)$$

$$(96) \wedge (99) \wedge (105) \wedge (112) \implies P_6^1 \text{ is final-state last-use opaque} \quad (113)$$

□

Let  $P_7^1$  be a prefix s.t.  $H_1 = P_7^1 \cdot [\text{res}_i(ok_i), r_j(x) \rightarrow 1, \text{try}C_i \rightarrow C_i, \text{try}C_j \rightarrow C_j]$ .

**Lemma 19.**  $P_7^1$  is final-state last-use opaque.

*Proof.*

$$\text{let } C_7^1 = \text{Compl}(P_7^1) = P_7^1 \cdot [\text{res}_i(A_i), \text{try}A_j \rightarrow A_j] \quad (114)$$

$$\text{let } S_7^1 = C_7^1|T_i \cdot C_7^1|T_j \quad (115)$$

$$S_7^1 \equiv C_7^1 \quad (116)$$

$$\text{real time order } <_{P_7^1} = \emptyset \quad (117)$$

$$\text{real time order } <_{S_7^1} = \{T_i <_{S_7^1} T_j\} \quad (118)$$

$$(117) \wedge (118) \implies <_{S_7^1} \subseteq <_{P_7^1} \quad (119)$$

$$i = i \implies S_7^1|T_i \subseteq \text{LVis}(S_7^1, T_i) \quad (120)$$

$$T_i <_{S_7^1} T_j \implies S_7^1|T_j \not\subseteq \text{LVis}(S_7^1, T_i) \quad (121)$$

$$(120) \wedge (121) \implies \text{LVis}(S_7^1, T_i) = S_7^1|T_i \quad (122)$$

$$(122) \implies \text{Vis}(S_7^1, T_i)|x = [w_i(x)1 \rightarrow A_i] \quad (123)$$

$$(123) \wedge (2) \implies \text{LVis}(S_7^1, T_i) \text{ is legal} \quad (124)$$

$$(124) \implies T_i \text{ in } S_7^1 \text{ is last-use legal in } S_7^1 \quad (125)$$

$$w_i(x)1 \rightarrow A_i \text{ is not closing write on } x \text{ in } T_i \quad (126)$$

$$S_7^1|T_j|x \setminus \{w_i(x)1 \rightarrow A_i\} = \emptyset \quad (127)$$

$$\text{res}_i(C_i) \notin S_7^1|T_i \wedge (127) \implies S_7^1|T_i \not\subseteq \text{LVis}(S_7^1, T_j) \quad (128)$$

$$j = j \implies S_7^1|T_j \subseteq \text{LVis}(S_7^1, T_j) \quad (129)$$

$$(128) \wedge (129) \implies \text{LVis}(S_7^1, T_j) = S_7^1|T_j \quad (130)$$

$$(130) \implies \text{LVis}(S_7^1, T_j)|x = \square \quad (131)$$

$$(131) \wedge (8) \implies \text{LVis}(S_7^1, T_j) \text{ is legal} \quad (132)$$

$$(132) \implies T_j \text{ in } S_7^1 \text{ is last-use legal in } S_7^1 \quad (133)$$

$$(116) \wedge (119) \wedge (125) \wedge (133) \implies P_7^1 \text{ is final-state last-use opaque} \quad (134)$$

□

Let  $P_p^1$  be a any prefix s.t.  $H_1 = P_p^1 \cdot R \cdot [w_i(x)1 \rightarrow ok_i, r_j(x) \rightarrow 1, \text{try}C_i \rightarrow C_i, \text{try}C_j \rightarrow C_j]$ .

**Lemma 20.** *Any  $P_p^1$  is final-state last-use opaque.*

*Proof.* Since  $P_p^1$  does not contain any reads or writes, for any sequential history  $S \equiv P_p^1$ , transactions  $T_i$  and  $T_j$  are trivially both legal and last-use legal in  $S$ . Thus,  $P_p^1$  is final-state last-use opaque. □

**Lemma 21.**  *$H_1$  is last-use opaque.*

*Proof.* Since, from Lemmas 35–20, all prefixes of  $H_1$  are final-state last-use opaque, then by Def. 22  $H_1$  is last-use opaque. □

**Corollary 8.** *Any prefix of  $H_1$  is last-use opaque.*

**Lemma 22.**  *$H_2$  is final-state last-use opaque.*

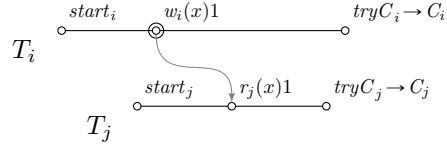


Figure 29: History  $H_2$ , not last-use opaque.

*Proof.*

$$\text{let } C_2 = \text{Compl}(H_2) = H_2 \quad (135)$$

$$\text{let } S_2 = C_1 | T_i \cdot C_2 | T_j \quad (136)$$

$$S_2 \equiv C_2 \quad (137)$$

$$S_2 = S_1 \wedge \text{Lemma 12} \implies H_2 \text{ is final-state last-use opaque} \quad (138)$$

□

Let  $P_1^2$  be a prefix s.t.  $H_2 = P_1^2 \cdot [\text{res}_i(C_i)]$ .

**Lemma 23.**  $P_1^2$  is final-state last-use opaque.

*Proof.*

$$\text{let } C_1^2 = \text{Compl}(P_1^2) = P_1^2 \cdot [\text{res}_j(C_j)] \quad (139)$$

$$H_2 = C_1^2 \wedge \text{Lemma 22} \implies P_1^2 \text{ is final-state last-use opaque} \quad (140)$$

□

Let  $P_2^2$  be a prefix s.t.  $H_2 = P_2^2 \cdot [\text{try} C_i \rightarrow C_i]$ .

**Lemma 24.**  $P_2^2$  is not final-state last-use opaque.

*Proof.*

$$\text{let } C_2^2 = \text{Compl}(P_2^2) = P_2^2 \cdot [\text{try} A_j \rightarrow A_j] \quad (141)$$

$$\text{let } S_2^2 = C_2^2|T_i \cdot C_2^2|T_j \quad (142)$$

$$S_2^2 \equiv C_2^2 \quad (143)$$

$$\text{real time order } <_{P_2^2} = \emptyset \quad (144)$$

$$\text{real time order } <_{S_2^2} = \{T_i <_{S_2^2} T_j\} \quad (145)$$

$$(144) \wedge (145) \implies <_{S_2^2} \subseteq <_{P_2^2} \quad (146)$$

$$\text{res}_i(A_i) \in S_2^2|T_i \implies S_2^2|T_i \not\subseteq \text{LVis}(S_2^2, T_j) \quad (147)$$

$$j = j \implies S_2^2|T_j \subseteq \text{LVis}(S_2^2, T_j) \quad (148)$$

$$(147) \wedge (148) \implies \text{LVis}(S_2^2, T_j) = S_2^2|T_j \quad (149)$$

$$(149) \implies \text{LVis}(S_2^2, T_j)|x = [r_j(x) \rightarrow 1] \quad (150)$$

$$(150) \wedge (9) \implies \text{LVis}(S_2^2, T_j) \text{ is not legal} \quad (151)$$

$$(151) \implies T_j \text{ in } S_2^2 \text{ is not legal in } S_2^2 \quad (152)$$

$$\text{let } \dot{S}_2^2 = C_2^2|T_j \cdot C_2^2|T_i \quad (153)$$

$$\dot{S}_2^2 \equiv C_2^2 \quad (154)$$

$$\text{real time order } <_{P_2^2} = \emptyset \quad (155)$$

$$\text{real time order } <_{\dot{S}_2^2} = \{T_i <_{\dot{S}_2^2} T_j\} \quad (156)$$

$$(155) \wedge (156) \implies <_{\dot{S}_2^2} \subseteq <_{P_2^2} \quad (157)$$

$$T_j <_{\dot{S}_2^2} T_i \implies \dot{S}_2^2|T_i \not\subseteq \text{Vis}(\dot{S}_2^2, T_j) \quad (158)$$

$$j = j \implies \dot{S}_2^2|T_j \subseteq \text{Vis}(\dot{S}_2^2, T_i) \quad (159)$$

$$(158) \wedge (159) \implies \text{Vis}(\dot{S}_2^2, T_j) = \dot{S}_2^2|T_j \quad (160)$$

$$(160) \implies \text{Vis}(\dot{S}_2^2, T_j)|x = [r_j(x) \rightarrow 1] \quad (161)$$

$$(161) \wedge (9) \implies \text{LVis}(\dot{S}_2^2, T_j) \text{ is not legal} \quad (162)$$

$$(162) \implies T_j \text{ in } \dot{S}_2^2 \text{ is not legal in } \dot{S}_2^2 \quad (163)$$

$$(152) \wedge (163) \implies P_2^2 \text{ is not final-state last-use opaque} \quad (164)$$

□

**Lemma 25.**  $H_2$  is not last-use opaque.

*Proof.* Even though, from Lemma 22,  $H_2$  is final-state last-use opaque, from Lemma 24, prefix  $P_2^2$  of  $H_2$  is not final-state last-use opaque, so, from Def. 22  $H_2$  is not last-use opaque. □

**Lemma 26.**  $H_3$  is final-state last-use opaque.

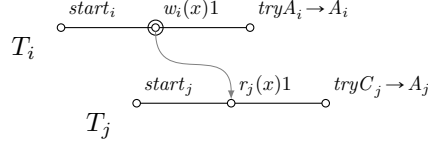


Figure 30: History  $H_3$ , last-use opaque.

*Proof.*

$$\text{let } C_3 = \text{Compl}(H_3) = H_3 \quad (165)$$

$$\text{let } S_3 = C_3|T_i \cdot C_3|T_j \quad (166)$$

$$S_3 \equiv C_3 \quad (167)$$

$$\text{real time order } <_{H_3} = \emptyset \quad (168)$$

$$\text{real time order } <_{S_3} = \{T_i <_{S_3} T_j\} \quad (169)$$

$$(168) \wedge (169) \implies <_{S_3} \subseteq <_{H_3} \quad (170)$$

$$i = i \implies S_3|T_i \subseteq \text{LVis}(S_3, T_i) \quad (171)$$

$$T_i <_{S_3} T_j \implies S_3|T_j \not\subseteq \text{LVis}(S_3, T_i) \quad (172)$$

$$(171) \wedge (172) \implies \text{LVis}(S_3, T_i) = S_3|T_i \quad (173)$$

$$(173) \implies \text{Vis}(S_3, T_i)|x = [w_i(x)1 \rightarrow ok_i] \quad (174)$$

$$(174) \wedge (2) \implies \text{LVis}(S_3, T_i) \text{ is legal} \quad (175)$$

$$(175) \implies T_i \text{ in } S_3 \text{ is last-use legal in } S_3 \quad (176)$$

$$w_i(x)1 \rightarrow ok_i \text{ is closing write on } x \text{ in } T_i \implies T_i \text{ is decided on } xS_3 \quad (177)$$

$$T_i <_{S_3} T_j \wedge (177) \implies S_3|\overset{\circ}{T_i} \subseteq \text{LVis}(S_3, T_j) \quad (178)$$

$$j = j \implies S_3|T_j \subseteq \text{LVis}(S_3, T_j) \quad (179)$$

$$(178) \wedge (179) \implies \text{LVis}(S_3, T_j) = S_3|\overset{\circ}{T_i} \cdot S_3|T_j \quad (180)$$

$$(180) \implies \text{LVis}(S_3, T_j)|x = [w_i(x)1 \rightarrow ok_i, r_j(x) \rightarrow 1] \quad (181)$$

$$(181) \wedge (4) \implies \text{LVis}(S_3, T_j) \text{ is legal} \quad (182)$$

$$(182) \implies T_j \text{ in } S_3 \text{ is last-use legal in } S_3 \quad (183)$$

$$(167) \wedge (170) \wedge (176) \wedge (183) \implies H_3 \text{ is final-state last-use opaque} \quad (184)$$

□

Let  $P_1^3$  be a prefix s.t.  $H_3 = P_1^3 \cdot [\text{res}_j(A_j)]$ .

**Lemma 27.**  $P_1^3$  is final-state last-use opaque.

*Proof.*

$$\text{let } C_1^3 = \text{Compl}(P_1^3) = P_1^3 \cdot [\text{res}_j(A_j)] \quad (185)$$

$$H_3 = C_1^3 \wedge \text{Lemma 26} \implies P_1^3 \text{ is final-state last-use opaque} \quad (186)$$

□

Let  $P_2^3$  be a prefix s.t.  $H_3 = P_2^3 \cdot [tryC_j \rightarrow A_j]$ .

**Lemma 28.**  $P_2^3$  is final-state last-use opaque.

*Proof.*

$$\text{let } C_2^3 = Compl(P_2^3) = P_2^3 \cdot [tryA_j \rightarrow A_j] \quad (187)$$

$$\text{let } S_2^3 = C_2^3|T_i \cdot C_2^3|T_j \quad (188)$$

$$S_2^3 \equiv C_2^3 \quad (189)$$

$$\text{real time order } <_{P_2^3} = \emptyset \quad (190)$$

$$\text{real time order } <_{S_2^3} = \{T_i <_{S_2^3} T_j\} \quad (191)$$

$$(190) \wedge (191) \implies <_{S_2^3} \subseteq <_{P_2^3} \quad (192)$$

$$i = i \implies S_2^3|T_i \subseteq LVis(S_2^3, T_i) \quad (193)$$

$$T_i <_{S_2^3} T_j \implies S_2^3|T_j \not\subseteq LVis(S_2^3, T_i) \quad (194)$$

$$(193) \wedge (194) \implies LVis(S_2^3, T_i) = S_2^3|T_i \quad (195)$$

$$(195) \implies Vis(S_2^3, T_i)|x = [w_i(x)1 \rightarrow ok_i] \quad (196)$$

$$(196) \wedge (2) \implies LVis(S_2^3, T_i) \text{ is legal} \quad (197)$$

$$(197) \implies T_i \text{ in } S_2^3 \text{ is last-use legal in } S_2^3 \quad (198)$$

$$w_i(x)1 \rightarrow ok_i \text{ is closing write on } x \text{ in } T_i \implies T_i \text{ is decided on } xS_2^3 \quad (199)$$

$$T_i <_{S_2^3} T_j \wedge (199) \implies S_2^3|T_i \subseteq LVis(S_2^3, T_j) \quad (200)$$

$$j = j \implies S_2^3|T_j \subseteq LVis(S_2^3, T_j) \quad (201)$$

$$(200) \wedge (201) \implies LVis(S_2^3, T_j) = S_2^3|T_i \cdot S_2^3|T_j \quad (202)$$

$$(202) \implies LVis(S_2^3, T_j)|x = [w_i(x)1 \rightarrow ok_i, r_j(x) \rightarrow 1] \quad (203)$$

$$(203) \wedge (4) \implies LVis(S_2^3, T_j) \text{ is legal} \quad (204)$$

$$(204) \implies T_j \text{ in } S_2^3 \text{ is last-use legal in } S_2^3 \quad (205)$$

$$(189) \wedge (192) \wedge (198) \wedge (205) \implies P_2^3 \text{ is final-state last-use opaque} \quad (206)$$

□

Let  $P_3^3$  be a prefix s.t.  $H_3 = P_3^3 \cdot [tryC_j \rightarrow A_j]$ .

**Lemma 29.**  $P_3^3$  is final-state last-use opaque.

*Proof.*

$$P_3^3 = P_4^1 \wedge \text{Lemma 16} \implies P_3^3 \text{ is final-state last-use opaque} \quad (207)$$

□

Let  $P_p^3$  be a any prefix s.t.  $P_3^3$ .

**Lemma 30.** Any  $P_p^3$  is final-state last-use opaque.

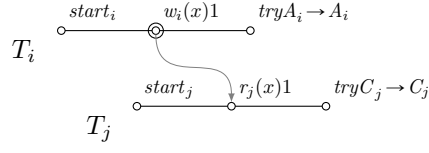


Figure 31: History  $H_4$ , not last-use opaque.

*Proof.*

$$H_1 = P_4^1 \cdot R \wedge \text{Corollary 9} \implies P_4^1 \text{ is last-use opaque} \quad (208)$$

$$P_4^1 = P_3^3 \wedge (208) \implies P_3^3 \text{ is last-use opaque} \quad (209)$$

$$(209) \implies P_3^3 \text{ is final-state last-use opaque} \quad (210)$$

□

**Lemma 31.**  $H_3$  is last-use opaque.

*Proof.* Since, from Lemmas 27–30, all prefixes of  $H_3$  are final-state last-use opaque, then by Def. 22  $H_3$  is last-use opaque. □

**Corollary 9.** Any prefix of  $H_3$  is last-use opaque.

**Lemma 32.**  $H_4$  is not final-state last-use opaque.

*Proof.*

$$\text{let } C_4 = \text{Compl}(H_4) = H_4 \quad (211)$$

$$\text{let } S_4 = C_4|T_i \cdot C_4|T_j \quad (212)$$

$$S_4 \equiv C_4 \quad (213)$$

$$\text{real time order } <_{P_4} = \emptyset \quad (214)$$

$$\text{real time order } <_{S_4} = \{T_i <_{S_4} T_j\} \quad (215)$$

$$(214) \wedge (215) \implies <_{S_4} \subseteq <_{P_4} \quad (216)$$

$$\text{res}_i(A_i) \in S_4|T_i \implies S_4|T_i \not\subseteq \text{Vis}(S_4, T_j) \quad (217)$$

$$j = j \implies S_4|T_j \subseteq \text{Vis}(S_4, T_j) \quad (218)$$

$$(217) \wedge (218) \implies \text{Vis}(S_4, T_j) = S_4|T_j \quad (219)$$

$$(219) \implies \text{Vis}(S_4, T_j)|x = [r_j(x) \rightarrow 1] \quad (220)$$

$$(220) \wedge (9) \implies \text{Vis}(S_4, T_j) \text{ is not legal} \quad (221)$$

$$(221) \implies T_j \text{ in } S_4 \text{ is not legal in } S_4 \quad (222)$$

$$\text{let } \dot{S}_4 = C_4|T_j \cdot C_4|T_i \quad (223)$$

$$\dot{S}_4 \equiv C_4 \quad (224)$$

$$\text{real time order } <_{P_4} = \emptyset \quad (225)$$

$$\text{real time order } <_{\dot{S}_4} = \{T_i <_{\dot{S}_4} T_j\} \quad (226)$$

$$(225) \wedge (226) \implies <_{\dot{S}_4} \subseteq <_{P_4} \quad (227)$$

$$T_j <_{\dot{S}_4} T_i \implies \dot{S}_4|T_i \not\subseteq \text{Vis}(\dot{S}_4, T_j) \quad (228)$$

$$j = j \implies \dot{S}_4|T_j \subseteq \text{Vis}(\dot{S}_4, T_j) \quad (229)$$

$$(228) \wedge (229) \implies \text{Vis}(\dot{S}_4, T_j) = \dot{S}_4|T_j \quad (230)$$

$$(230) \implies \text{Vis}(\dot{S}_4, T_j)|x = [r_j(x) \rightarrow 1] \quad (231)$$

$$(231) \wedge (9) \implies \text{Vis}(\dot{S}_4, T_j) \text{ is not legal} \quad (232)$$

$$(232) \implies T_j \text{ in } \dot{S}_4 \text{ is not legal in } \dot{S}_4 \quad (233)$$

$$(222) \wedge (233) \implies P_4 \text{ is not final-state last-use opaque} \quad (234)$$

□

**Lemma 33.**  $H_4$  is not last-use opaque.

*Proof.* From Lemma 32 is not final-state last-use opaque, then so, from Def. 22  $H_4$  is not last-use opaque. □

**Lemma 34.**  $H_5$  is final-state last-use opaque.

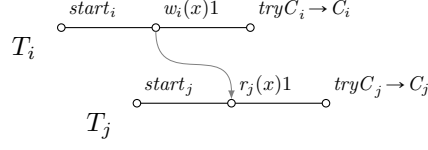


Figure 32: History  $H_5$ , not last-use opaque. Note that write in  $T_i$  is not closing write.

*Proof.*

$$\text{let } C_5 = \text{Compl}(H_5) = H_5 \quad (235)$$

$$\text{let } S_5 = C_5|T_i \cdot C_5|T_j \quad (236)$$

$$S_5 \equiv C_5 \quad (237)$$

$$\text{real time order } <_{H_5} = \emptyset \quad (238)$$

$$\text{real time order } <_{S_5} = \{T_i <_{S_5} T_j\} \quad (239)$$

$$(238) \wedge (239) \implies <_{S_5} \subseteq <_{H_5} \quad (240)$$

$$S_5 = S_1 \wedge (237) \wedge (240) \wedge (23) \wedge (29) \implies H_5 \text{ is final-state last-use opaque} \quad (241)$$

□

Let  $P_1^5$  be a prefix s.t.  $H_5 = P_1^5 \cdot [\text{res}_j(C_j)]$ .

**Lemma 35.**  $P_1^5$  is final-state last-use opaque.

*Proof.*

$$\text{let } C_1^5 = \text{Compl}(P_1^5) = P_1^5 \cdot [\text{res}_j(C_j)] \quad (242)$$

$$H_5 = C_1^5 \wedge \text{Lemma 12} \implies P_1^5 \text{ is final-state last-use opaque} \quad (243)$$

□

Let  $P_2^5$  be a prefix s.t.  $H_5 = P_2^5 \cdot [\text{try} C_j \rightarrow C_j]$ .

**Lemma 36.**  $P_2^5$  is final-state last-use opaque.

*Proof.*

$$\text{let } C_2^5 = \text{Compl}(P_2^5) = P_2^5 \cdot [\text{try}A_j \rightarrow A_j] \quad (244)$$

$$\text{let } S_2^5 = C_2^5|T_i \cdot C_2^5|T_j \quad (245)$$

$$S_2^5 \equiv C_2^5 \quad (246)$$

$$\text{real time order } <_{P_2^5} = \emptyset \quad (247)$$

$$\text{real time order } <_{S_2^5} = \{T_i <_{S_2^5} T_j\} \quad (248)$$

$$(247) \wedge (248) \implies <_{S_2^5} \subseteq <_{P_2^5} \quad (249)$$

$$i = i \implies S_2^5|T_i \subseteq \text{Vis}(S_2^5, T_i) \quad (250)$$

$$T_i <_{S_2^5} T_j \implies S_2^5|T_j \not\subseteq \text{Vis}(S_2^5, T_i) \quad (251)$$

$$(250) \wedge (251) \implies \text{Vis}(S_2^5, T_i) = S_2^5|T_i \quad (252)$$

$$(252) \implies \text{Vis}(S_2^5, T_i)|x = [w_i(x)1 \rightarrow ok_i] \quad (253)$$

$$(253) \wedge (2) \implies \text{Vis}(S_2^5, T_i) \text{ is legal} \quad (254)$$

$$(254) \implies T_i \text{ in } S_2^5 \text{ is legal in } S_2^5 \quad (255)$$

$$T_i <_{S_2^5} T_j \wedge \text{res}_i(C_i) \in S_2^5|T_i \implies S_2^5|T_i \subseteq \text{LVis}(S_2^5, T_j) \quad (256)$$

$$j = j \implies S_2^5|T_j \subseteq \text{LVis}(S_2^5, T_j) \quad (257)$$

$$(256) \wedge (257) \implies \text{LVis}(S_2^5, T_j) = S_2^5|T_i \cdot S_2^5|T_j \quad (258)$$

$$(258) \implies \text{LVis}(S_2^5, T_j)|x = [w_i(x)1 \rightarrow ok_i, r_j(x) \rightarrow 1] \quad (259)$$

$$(259) \wedge (4) \implies \text{LVis}(S_2^5, T_j) \text{ is legal} \quad (260)$$

$$(260) \implies T_j \text{ in } S_2^5 \text{ is last-use legal in } S_2^5 \quad (261)$$

$$(246) \wedge (249) \wedge (255) \wedge (261) \implies P_2^5 \text{ is final-state last-use opaque} \quad (262)$$

□

Let  $P_3^5$  be a prefix s.t.  $H_5 = P_3^5 \cdot [\text{res}_i(C_i), \text{try}C_j \rightarrow C_j]$ .

**Lemma 37.**  $P_3^5$  is final-state last-use opaque.

*Proof.*

$$\text{let } C_3^5 = \text{Compl}(P_3^5) = P_3^5 \cdot [\text{res}_i(C_i), \text{try}C_j \rightarrow C_j] \quad (263)$$

$$P_2^5 = C_3^5 \wedge \text{Lemma 37} \implies P_3^5 \text{ is final-state last-use opaque} \quad (264)$$

□

Let  $P_4^5$  be a prefix s.t.  $H_5 = P_5^1 \cdot [\text{try}C_i \rightarrow C_i, \text{try}C_j \rightarrow C_j]$ .

**Lemma 38.**  $P_4^5$  is not final-state last-use opaque.

*Proof.*

$$\text{let } C_4^5 = \text{Compl}(P_4^5) = P_4^5 \cdot [\text{try}A_i \rightarrow A_i, \text{try}A_j \rightarrow A_j] \quad (265)$$

$$\text{let } S_4^5 = C_4^5|T_i \cdot C_4^5|T_j \quad (266)$$

$$S_4^5 \equiv C_4^5 \quad (267)$$

$$\text{real time order } <_{P_4^5} = \emptyset \quad (268)$$

$$\text{real time order } <_{S_4^5} = \{T_i <_{S_4^5} T_j\} \quad (269)$$

$$(268) \wedge (269) \implies <_{S_4^5} \subseteq <_{P_4^5} \quad (270)$$

$$w_i(x)1 \rightarrow ok_i \text{ is not closing write on } x \text{ in } T_i \implies T_i \text{ is not decided on } x \quad (271)$$

$$T_i <_{S_4^5} T_j \wedge (271) \implies S_4^5|T_i \not\subseteq \text{LVis}(S_4^5, T_j) \quad (272)$$

$$j = j \implies S_4^5|T_j \subseteq \text{LVis}(S_4^5, T_j) \quad (273)$$

$$(272) \wedge (273) \implies \text{LVis}(S_4^5, T_j) = S_4^5|T_i \cdot S_4^5|T_j \quad (274)$$

$$(274) \implies \text{LVis}(S_4^5, T_j)|x = [r_j(x) \rightarrow 1] \quad (275)$$

$$(275) \wedge (4) \implies \text{LVis}(S_4^5, T_j) \text{ is not legal} \quad (276)$$

$$(276) \implies T_j \text{ in } S_4^5 \text{ is not last-use legal in } S_4^5 \quad (277)$$

$$\text{let } \dot{S}_4^5 = C_4^5|T_j \cdot C_4^5|T_i \quad (278)$$

$$\dot{S}_4^5 \equiv C_4^5 \quad (279)$$

$$\text{real time order } <_{P_4^5} = \emptyset \quad (280)$$

$$\text{real time order } <_{\dot{S}_4^5} = \{T_i <_{\dot{S}_4^5} T_j\} \quad (281)$$

$$(280) \wedge (281) \implies <_{\dot{S}_4^5} \subseteq <_{P_4^5} \quad (282)$$

$$T_j <_{\dot{S}_4^5} T_i \implies \dot{S}_4^5|T_i \not\subseteq \text{LVis}(\dot{S}_4^5, T_j) \quad (283)$$

$$j = j \implies \dot{S}_4^5|T_j \subseteq \text{LVis}(\dot{S}_4^5, T_i) \quad (284)$$

$$(283) \wedge (284) \implies \text{LVis}(\dot{S}_4^5, T_j) = \dot{S}_4^5|T_j \quad (285)$$

$$(285) \implies \text{LVis}(\dot{S}_4^5, T_j)|x = [r_j(x) \rightarrow 1] \quad (286)$$

$$(286) \wedge (9) \implies \text{LVis}(\dot{S}_4^5, T_j) \text{ is not legal} \quad (287)$$

$$(287) \implies T_j \text{ in } \dot{S}_4^5 \text{ is not legal in } \dot{S}_4^5 \quad (288)$$

$$(277) \wedge (288) \implies P_2^2 \text{ is not final-state last-use opaque} \quad (289)$$

□

**Lemma 39.**  $H_5$  is not last-use opaque.

*Proof.* Even though, from Lemma 34,  $H_5$  is final-state last-use opaque, from Lemma 38, prefix  $P_4^5$  of  $H_5$  is not final-state last-use opaque, so, from Def. 22  $H_5$  is not last-use opaque. □

**Lemma 40.**  $H_6$  is not final-state last-use opaque.

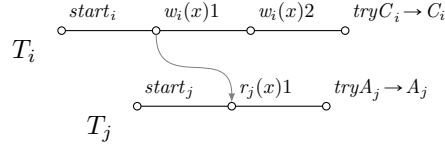


Figure 33: History  $H_6$ , not last-use opaque.

*Proof.*

$$\text{let } C_6 = \text{Compl}(H_6) = H_6 \quad (290)$$

$$\text{let } S_6 = C_6|T_i \cdot C_6|T_j \quad (291)$$

$$S_6 \equiv C_6 \quad (292)$$

$$\text{real time order } <_{H_6} = \emptyset \quad (293)$$

$$\text{real time order } <_{S_6} = \{T_i <_{S_6} T_j\} \quad (294)$$

$$(293) \wedge (294) \implies <_{S_6} \subseteq <_{H_6} \quad (295)$$

$$T_i <_{S_6} T_j \wedge \text{res}_i(C_i) \in S_6|T_i \implies S_6|T_i \subseteq \text{LVis}(S_6, T_i) \quad (296)$$

$$j = j \implies S_6|T_j \subseteq \text{LVis}(S_6, T_i) \quad (297)$$

$$(296) \wedge (297) \implies \text{Vis}(S_6, T_j) = S_6|T_i \cdot S_6|T_j \quad (298)$$

$$(298) \implies \text{Vis}(S_6, T_j)|x = [w_i(x)1 \rightarrow \text{ok}_i, w_i(x)2 \rightarrow \text{ok}_i, r_j(x) \rightarrow 1] \quad (299)$$

$$(299) \wedge (10) \implies \text{Vis}(S_6, T_j) \text{ is not legal} \quad (300)$$

$$(300) \implies T_j \text{ in } S_6 \text{ is not legal in } S_6 \quad (301)$$

$$\text{let } \dot{S}_6 = C_6|T_j \cdot C_6|T_i \quad (302)$$

$$\dot{S}_6 \equiv C_6 \quad (303)$$

$$\text{real time order } <_{H_6} = \emptyset \quad (304)$$

$$\text{real time order } <_{\dot{S}_6} = \{T_i <_{\dot{S}_6} T_j\} \quad (305)$$

$$(304) \wedge (305) \implies <_{\dot{S}_6} \subseteq <_{H_6} \quad (306)$$

$$T_j <_{\dot{S}_6} T_i \implies \dot{S}_6|T_i \not\subseteq \text{LVis}(\dot{S}_6, T_j) \quad (307)$$

$$j = j \implies \dot{S}_6|T_j \subseteq \text{LVis}(\dot{S}_6, T_i) \quad (308)$$

$$(307) \wedge (308) \implies \text{LVis}(\dot{S}_6, T_j) = \dot{S}_6|T_j \quad (309)$$

$$(309) \implies \text{LVis}(\dot{S}_6, T_j)|x = [r_j(x) \rightarrow 1] \quad (310)$$

$$(310) \wedge (9) \implies \text{LVis}(\dot{S}_6, T_j) \text{ is not legal} \quad (311)$$

$$(311) \implies T_j \text{ in } \dot{S}_6 \text{ is not legal in } \dot{S}_6 \quad (312)$$

$$(301) \wedge (312) \implies H_6 \text{ is not final-state last-use opaque} \quad (313)$$

□

**Lemma 41.**  $H_6$  is not last-use opaque.

*Proof.* From Lemma 40 is not final-state last-use opaque, then so, from Def. 22  $H_6$  is not last-use opaque. □

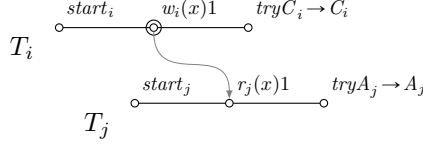


Figure 34: History  $H_7$ , last-use opaque.

**Lemma 42.**  $H_7$  is final-state last-use opaque.

*Proof.*

$$\text{let } C_7 = \text{Compl}(H_7) = H_7 \quad (314)$$

$$\text{let } S_7 = C_7|T_i \cdot C_7|T_j \quad (315)$$

$$S_7 \equiv C_7 \quad (316)$$

$$\text{real time order } <_{H_7} = \emptyset \quad (317)$$

$$\text{real time order } <_{S_7} = \{T_i <_{S_7} T_j\} \quad (318)$$

$$(317) \wedge (318) \implies <_{S_7} \subseteq <_{H_7} \quad (319)$$

$$i = i \implies S_7|T_i \subseteq \text{Vis}(S_7, T_i) \quad (320)$$

$$T_i <_{S_7} T_j \implies S_7|T_j \not\subseteq \text{Vis}(S_7, T_i) \quad (321)$$

$$(320) \wedge (321) \implies \text{Vis}(S_7, T_i) = S_7|T_i \quad (322)$$

$$(322) \implies \text{Vis}(S_7, T_i)|x = [w_i(x)1 \rightarrow ok_i] \quad (323)$$

$$(323) \wedge (2) \implies L\text{Vis}(S_7, T_i) \text{ is legal} \quad (324)$$

$$(324) \implies T_i \text{ in } S_7 \text{ is last-use legal in } S_7 \quad (325)$$

$$T_i <_{S_7} T_j \wedge \text{res}_i(C_i) \in S_7|T_i \implies S_7|T_i \subseteq \text{Vis}(S_7, T_i) \quad (326)$$

$$j = j \implies S_7|T_j \subseteq \text{Vis}(S_7, T_j) \quad (327)$$

$$(326) \wedge (327) \implies \text{Vis}(S_7, T_j) = S_7|T_i \cdot S_7|T_j \quad (328)$$

$$(328) \implies L\text{Vis}(S_7, T_j)|x = [w_i(x)1 \rightarrow ok_i, r_j(x) \rightarrow 1] \quad (329)$$

$$(329) \wedge (4) \implies L\text{Vis}(S_7, T_j) \text{ is legal} \quad (330)$$

$$(330) \implies T_j \text{ in } S_7 \text{ is last-use legal in } S_7 \quad (331)$$

$$(316) \wedge (319) \wedge (325) \wedge (331) \implies H_7 \text{ is final-state last-use opaque} \quad (332)$$

□

Let  $P_1^7$  be a prefix s.t.  $H_7 = P_1^7 \cdot [\text{res}_i(C_i)]$ .

**Lemma 43.**  $P_1^7$  is final-state last-use opaque.

*Proof.*

$$\text{let } C_1^7 = \text{Compl}(P_1^7) = P_1^7 \cdot [\text{res}_i(C_i)] \quad (333)$$

$$H_7 = C_1^7 \wedge \text{Lemma 42} \implies P_1^7 \text{ is final-state last-use opaque} \quad (334)$$

□

Let  $P_2^7$  be a prefix s.t.  $H_7 = P_2^7 \cdot [tryC_i \rightarrow C_i]$ .

*Proof.*

$$\text{let } C_2^7 = Compl(P_2^7) = P_2^7 \cdot [tryA_i \rightarrow A_i] \quad (335)$$

$$\text{let } S_2^7 = C_2^7|T_i \cdot C_2^7|T_j \quad (336)$$

$$S_2^7 \equiv C_2^7 \quad (337)$$

$$\text{real time order } <_{P_2^7} = \emptyset \quad (338)$$

$$\text{real time order } <_{S_2^7} = \{T_i <_{S_2^7} T_j\} \quad (339)$$

$$(338) \wedge (339) \implies <_{S_2^7} \subseteq <_{P_2^7} \quad (340)$$

$$i = i \implies S_2^7|T_i \subseteq LVis(S_2^7, T_i) \quad (341)$$

$$T_i <_{S_2^7} T_j \implies S_2^7|T_j \not\subseteq LVis(S_2^7, T_i) \quad (342)$$

$$(341) \wedge (342) \implies LVis(S_2^7, T_i) = S_2^7|T_i \quad (343)$$

$$(343) \implies Vis(S_2^7, T_i)|x = [w_i(x)1 \rightarrow ok_i] \quad (344)$$

$$(344) \wedge (2) \implies LVis(S_2^7, T_i) \text{ is legal} \quad (345)$$

$$(345) \implies T_i \text{ in } S_2^7 \text{ is last-use legal in } S_2^7 \quad (346)$$

$$w_i(x)1 \rightarrow ok_i \text{ is closing write on } x \text{ in } T_i \implies T_i \text{ is decided on } x \quad (347)$$

$$T_i <_{S_2^7} T_j \wedge (347) \implies S_2^7|T_i \subseteq LVis(S_2^7, T_j) \quad (348)$$

$$j = j \implies S_2^7|T_j \subseteq LVis(S_2^7, T_j) \quad (349)$$

$$(348) \wedge (349) \implies LVis(S_2^7, T_j) = S_2^7|T_i \cdot S_2^7|T_j \quad (350)$$

$$(350) \implies LVis(S_2^7, T_j)|x = [w_i(x)1 \rightarrow ok_i] \quad (351)$$

$$(351) \wedge (2) \implies LVis(S_2^7, T_j) \text{ is legal} \quad (352)$$

$$(352) \implies T_j \text{ in } S_2^7 \text{ is last-use legal in } S_2^7 \quad (353)$$

$$(337) \wedge (340) \wedge (346) \wedge (353) \implies P_2^7 \text{ is final-state last-use opaque} \quad (354)$$

□

**Lemma 44.**  $P_3^7$  is final-state last-use opaque.

*Proof.*

$$\text{let } C_3^7 = Compl(P_3^7) = P_3^7 \cdot [res_j(A_j), tryC_i \rightarrow C_i] \quad (355)$$

$$P_3^7 = C_3^7 \wedge \text{Lemma 44} \implies P_3^7 \text{ is final-state last-use opaque} \quad (356)$$

□

Let  $P_4^7$  be a prefix s.t.  $H_3 = P_4^7 \cdot [tryC_j \rightarrow A_j, tryC_i \rightarrow C_i]$ .

**Lemma 45.**  $P_4^7$  is final-state last-use opaque.

*Proof.*

$$P_4^7 = P_4^5 \wedge \text{Lemma 38} \implies P_4^7 \text{ is final-state last-use opaque} \quad (357)$$

□

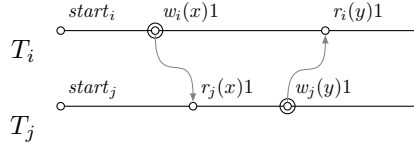


Figure 35: History  $H_8$ , not last-use opaque.

Let  $P_p^7$  be a any prefix of  $P_4^7$ .

**Lemma 46.** *Any  $P_p^7$  is final-state last-use opaque.*

*Proof.*

$$H_1 = P_4^1 \cdot R \wedge \text{Corollary 8} \implies P_4^1 \text{ is last-use lopaque} \quad (358)$$

$$P_4^1 = P_4^7 \wedge (358) \implies P_4^7 \text{ is last-use lopaque} \quad (359)$$

$$(359) \implies P_p^7 \text{ is final-state last-use lopaque} \quad (360)$$

□

**Lemma 47.**  *$H_7$  is last-use opaque.*

*Proof.* Since, from Lemmas 43–46, all prefixes of  $H_7$  are final-state last-use opaque, then by Def. 22  $H_7$  is last-use opaque. □

**Lemma 48.**  *$H_8$  is not final-state last-use opaque.*

*Proof.*

$$\text{let } C_8 = \text{Compl}(H_8) = H_8 \quad (361)$$

$$\text{let } S_8 = C_8|T_i \cdot C_8|T_j \quad (362)$$

$$S_8 \equiv C_8 \quad (363)$$

$$\text{real time order } <_{H_8} = \emptyset \quad (364)$$

$$\text{real time order } <_{S_8} = \{T_i <_{S_8} T_j\} \quad (365)$$

$$(364) \wedge (365) \implies <_{S_8} \subseteq <_{H_8} \quad (366)$$

$$T_i <_{S_8} T_j \wedge \text{res}_i(C_i) \in S_8|T_i \implies S_8|T_i \subseteq \text{LVis}(S_8, T_i) \quad (367)$$

$$j = j \implies S_8|T_j \subseteq \text{LVis}(S_8, T_j) \quad (368)$$

$$(367) \wedge (368) \implies \text{Vis}(S_8, T_j) = S_8|T_i \cdot S_8|T_j \quad (369)$$

$$(369) \implies \text{Vis}(S_8, T_j)|x = [w_i(x)1 \rightarrow ok_i, r_i(y) \rightarrow 1, r_j(x) \rightarrow 1, w_j(y)1 \rightarrow ok_j] \quad (370)$$

$$(370) \wedge (11) \implies \text{Vis}(S_8, T_j) \text{ is not legal} \quad (371)$$

$$(371) \implies T_j \text{ in } S_8 \text{ is not legal in } S_8 \quad (372)$$

$$\text{let } \dot{S}_8 = C_8|T_j \cdot C_8|T_i \quad (373)$$

$$\dot{S}_8 \equiv C_8 \quad (374)$$

$$\text{real time order } <_{H_8} = \emptyset \quad (375)$$

$$\text{real time order } <_{\dot{S}_8} = \{T_i <_{\dot{S}_8} T_j\} \quad (376)$$

$$(375) \wedge (376) \implies <_{\dot{S}_8} \subseteq <_{H_8} \quad (377)$$

$$T_j <_{\dot{S}_8} T_i \implies \dot{S}_8|T_i \not\subseteq \text{LVis}(\dot{S}_8, T_j) \quad (378)$$

$$j = j \implies \dot{S}_8|T_j \subseteq \text{LVis}(\dot{S}_8, T_j) \quad (379)$$

$$(378) \wedge (379) \implies \text{LVis}(\dot{S}_8, T_j) = \dot{S}_8|T_j \quad (380)$$

$$(380) \implies \text{LVis}(\dot{S}_8, T_j)|x = [w_j(y)1 \rightarrow ok_i, r_j(x) \rightarrow 1, r_i(y) \rightarrow 1, w_i(x)1 \rightarrow ok_i] \quad (381)$$

$$(381) \wedge (11) \implies \text{LVis}(\dot{S}_8, T_j) \text{ is not legal} \quad (382)$$

$$(382) \implies T_j \text{ in } \dot{S}_8 \text{ is not legal in } \dot{S}_8 \quad (383)$$

$$(372) \wedge (383) \implies H_8 \text{ is not final-state last-use opaque} \quad (384)$$

□

**Lemma 49.**  $H_8$  is not last-use opaque.

*Proof.* From Lemma 48 is not final-state last-use opaque, then so, from Def. 22  $H_8$  is not last-use opaque. □

**Lemma 50.**  $H_9$  is final-state last-use opaque.

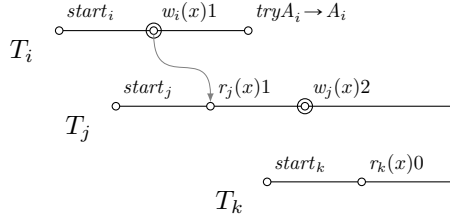


Figure 36:  $H_9$ , last-use opaque.

*Proof.*

$$\text{let } C_9 = \text{Compl}(H_9) = H^9 \cdot [\text{try}A_j \rightarrow A_j, \text{try}A_k \rightarrow A_k] \quad (385)$$

$$\text{let } S_9 = C_9|T_i \cdot C_9|T_j \cdot C_9|T_k \quad (386)$$

$$S_9 \equiv C_9 \quad (387)$$

$$\text{real time order } <_{H_9} = \{T_i <_{H_9} T_k\} \quad (388)$$

$$\text{real time order } <_{S_9} = \{T_i <_{S_9} T_j, T_i <_{S_9} T_k, T_j <_{S_9} T_k\} \quad (389)$$

$$(388) \wedge (389) \implies <_{S_9} \subseteq <_{H_9} \quad (390)$$

$$i = i \implies S_9|T_i \subseteq \text{LVis}(S_9, T_i) \quad (391)$$

$$T_i <_{S_9} T_j \implies S_9|T_j \not\subseteq \text{LVis}(S_9, T_i) \quad (392)$$

$$T_i <_{S_9} T_k \implies S_9|T_k \not\subseteq \text{LVis}(S_9, T_i) \quad (393)$$

$$(391) \wedge (392) \wedge (393) \implies \text{LVis}(S_9, T_i) = S_9|T_i \quad (394)$$

$$(394) \implies \text{LVis}(S_9, T_i)|x = [w_i(x)1 \rightarrow ok_i] \quad (395)$$

$$(395) \wedge (2) \implies \text{LVis}(S_9, T_i) \text{ is legal} \quad (396)$$

$$(396) \implies T_i \text{ in } S_9 \text{ is last-use legal in } S_9 \quad (397)$$

$$w_i(x)1 \rightarrow ok_i \text{ is closing write on } x \text{ in } T_i \implies T_i \text{ is decided on } x \text{ in } S_9 \quad (398)$$

$$T_i <_{S_9} T_j \wedge (398) \implies S_9|T_i \subseteq \text{LVis}(S_9, T_j) \quad (399)$$

$$j = j \implies S_9|T_j \subseteq \text{LVis}(S_9, T_j) \quad (400)$$

$$T_j <_{S_9} T_k \implies S_9|T_k \not\subseteq \text{LVis}(S_9, T_j) \quad (401)$$

$$(399) \wedge (400) \wedge (401) \implies \text{LVis}(S_9, T_j) = S_9|T_i \cdot S_9|T_j \quad (402)$$

$$(402) \implies \text{LVis}(S_9, T_j)|x = [w_i(x)1 \rightarrow ok_i, r_j(x) \rightarrow 1, w_j(x)2 \rightarrow ok_j] \quad (403)$$

$$(414) \wedge (6) \implies \text{LVis}(S_9, T_j) \text{ is legal} \quad (404)$$

$$(404) \implies T_j \text{ in } S_9 \text{ is last-use legal in } S_9 \quad (405)$$

$$(406)$$

$$T_i <_{S_9} T_k \wedge (398) \implies S_9|T_i \subseteq LVis(S_9, T_k) \text{ or } S_9|T_i \not\subseteq LVis(S_9, T_k) \quad (407)$$

$$(407) \implies S_9|T_i \not\subseteq LVis(S_9, T_k) \quad (408)$$

$$w_j(x)2 \rightarrow ok_j \text{ is closing write on } x \text{ in } T_j \implies T_j \text{ is decided on } x \text{ in } S_9 \quad (409)$$

$$T_j <_{S_9} T_k \wedge (409) \implies S_9|T_j \subseteq LVis(S_9, T_k) \text{ or } S_9|T_j \not\subseteq LVis(S_9, T_k) \quad (410)$$

$$(410) \implies S_9|T_j \not\subseteq LVis(S_9, T_k) \quad (411)$$

$$k = k \implies S_9|T_k \subseteq LVis(S_9, T_k) \quad (412)$$

$$(408) \wedge (411) \wedge (412) \implies LVis(S_9, T_k) = S_9|T_k \quad (413)$$

$$(413) \implies LVis(S_9, T_k)|x = [r_j(x) \rightarrow 0] \quad (414)$$

$$(414) \wedge (1) \implies LVis(S_9, T_k) \text{ is legal} \quad (415)$$

$$(415) \implies T_k \text{ in } S_9 \text{ is last-use legal in } S_9 \quad (416)$$

$$(387) \wedge (390) \wedge (397) \wedge (405) \wedge (416) \implies H_9 \text{ is final-state last-use opaque} \quad (417)$$

□

Let  $P_1^9$  be a prefix s.t.  $H_9 = P_1^9 \cdot [res_k(0)]$ .

**Lemma 51.**  $P_1^9$  is final-state last-use opaque.

*Proof.*

$$\text{let } C_1^9 = Compl(P_1^9) = P_1^9 \cdot [tryA_j \rightarrow A_j, res_k(A_k)] \quad (418)$$

$$\text{let } S_1^9 = C_1^9|T_i \cdot C_1^9|T_j \cdot C_1^9|T_k \quad (419)$$

$$S_1^9 \equiv C_1^9 \quad (420)$$

$$\text{real time order } <_{P_1^9} = \{T_i <_{P_1^9} T_k\} \quad (421)$$

$$\text{real time order } <_{S_1^9} = \{T_i <_{S_1^9} T_j, T_i <_{S_1^9} T_k, T_j <_{S_1^9} T_k\} \quad (422)$$

$$(421) \wedge (422) \implies <_{S_1^9} \subseteq <_{P_1^9} \quad (423)$$

$$i = i \implies S_1^9|T_i \subseteq LVis(S_1^9, T_i) \quad (424)$$

$$T_i <_{S_1^9} T_j \implies S_1^9|T_j \not\subseteq LVis(S_1^9, T_i) \quad (425)$$

$$T_i <_{S_1^9} T_k \implies S_1^9|T_k \not\subseteq LVis(S_1^9, T_i) \quad (426)$$

$$(424) \wedge (425) \wedge (426) \implies LVis(S_1^9, T_i) = S_1^9|T_i \quad (427)$$

$$(427) \implies LVis(S_1^9, T_i)|x = [w_i(x)1 \rightarrow ok_i] \quad (428)$$

$$(428) \wedge (2) \implies LVis(S_1^9, T_i) \text{ is legal} \quad (429)$$

$$(429) \implies T_i \text{ in } S_1^9 \text{ is last-use legal in } S_1^9 \quad (430)$$

$$(431)$$

$$w_i(x)1 \rightarrow ok_i \text{ is closing write on } x \text{ in } T_i \implies T_i \text{ is decided on } x \text{ in } S_1^9 \quad (432)$$

$$T_i <_{S_1^9} T_j \wedge (432) \implies S_1^9|T_i \subseteq LVis(S_1^9, T_j) \quad (433)$$

$$j = j \implies S_1^9|T_j \subseteq LVis(S_1^9, T_j) \quad (434)$$

$$T_j <_{S_1^9} T_k \implies S_1^9|T_k \not\subseteq LVis(S_1^9, T_j) \quad (435)$$

$$(433) \wedge (434) \wedge (435) \implies LVis(S_1^9, T_j) = S_1^9|T_i \cdot S_1^9|T_j \quad (436)$$

$$(436) \implies LVis(S_1^9, T_j)|x = [w_i(x)1 \rightarrow ok_i, r_j(x) \rightarrow 1, w_j(x)2 \rightarrow ok_j] \quad (437)$$

$$(447) \wedge (6) \implies LVis(S_1^9, T_j) \text{ is legal} \quad (438)$$

$$(438) \implies T_j \text{ in } S_1^9 \text{ is last-use legal in } S_1^9 \quad (439)$$

$$T_i <_{S_1^9} T_k \wedge (432) \implies S_1^9|T_i \subseteq LVis(S_1^9, T_k) \text{ or } S_1^9|T_i \not\subseteq LVis(S_1^9, T_k) \quad (440)$$

$$(440) \implies S_1^9|T_i \not\subseteq LVis(S_1^9, T_k) \quad (441)$$

$$w_j(x)2 \rightarrow ok_j \text{ is closing write on } x \text{ in } T_j \implies T_j \text{ is decided on } x \text{ in } S_1^9 \quad (442)$$

$$T_j <_{S_1^9} T_k \wedge (442) \implies S_1^9|T_j \subseteq LVis(S_1^9, T_k) \text{ or } S_1^9|T_j \not\subseteq LVis(S_1^9, T_k) \quad (443)$$

$$(443) \implies S_1^9|T_j \not\subseteq LVis(S_1^9, T_k) \quad (444)$$

$$k = k \implies S_1^9|T_k \subseteq LVis(S_1^9, T_k) \quad (445)$$

$$(441) \wedge (444) \wedge (445) \implies LVis(S_1^9, T_k) = S_1^9|T_k \quad (446)$$

$$(446) \implies LVis(S_1^9, T_k)|x = [r_k(x) \rightarrow A_k] \quad (447)$$

$$(447) \wedge (??) \implies LVis(S_1^9, T_k) \text{ is legal} \quad (448)$$

$$(448) \implies T_k \text{ in } S_1^9 \text{ is last-use legal in } S_1^9 \quad (449)$$

$$(420) \wedge (423) \wedge (430) \wedge (439) \wedge (449) \implies P_1^9 \text{ is final-state last-use opaque} \quad (450)$$

□

Let  $P_2^9$  be a prefix s.t.  $H_9 = P_2^9 \cdot [r_k(x) \rightarrow 0]$ .

*Proof.*

$$\text{let } C_2^9 = \text{Compl}(P_2^9) = P_2^9 \cdot [\text{res}_j(A_j), \text{res}_k(A_k)] \quad (451)$$

$$\text{let } S_2^9 = C_2^9|T_i \cdot C_2^9|T_j \cdot C_2^9|T_k \quad (452)$$

$$S_2^9 \equiv C_2^9 \quad (453)$$

$$\text{real time order } <_{P_2^9} = \{T_i <_{P_2^9} T_k\} \quad (454)$$

$$\text{real time order } <_{S_2^9} = \{T_i <_{S_2^9} T_j, T_i <_{S_2^9} T_k, T_j <_{S_2^9} T_k\} \quad (455)$$

$$(454) \wedge (455) \implies <_{S_2^9} \subseteq <_{P_2^9} \quad (456)$$

$$i = i \implies S_2^9|T_i \subseteq \text{LVis}(S_2^9, T_i) \quad (457)$$

$$T_i <_{S_2^9} T_j \implies S_2^9|T_j \not\subseteq \text{LVis}(S_2^9, T_i) \quad (458)$$

$$T_i <_{S_2^9} T_k \implies S_2^9|T_k \not\subseteq \text{LVis}(S_2^9, T_i) \quad (459)$$

$$(457) \wedge (458) \wedge (459) \implies \text{LVis}(S_2^9, T_i) = S_2^9|T_i \quad (460)$$

$$(460) \implies \text{LVis}(S_2^9, T_i)|x = [w_i(x)1 \rightarrow ok_i] \quad (461)$$

$$(461) \wedge (2) \implies \text{LVis}(S_2^9, T_i) \text{ is legal} \quad (462)$$

$$(462) \implies T_i \text{ in } S_2^9 \text{ is last-use legal in } S_2^9 \quad (463)$$

$$w_i(x)1 \rightarrow ok_i \text{ is closing write on } x \text{ in } T_i \implies T_i \text{ is decided on } x \text{ in } S_2^9 \quad (464)$$

$$T_i <_{S_2^9} T_j \wedge (464) \implies S_2^9|T_i \subseteq \text{LVis}(S_2^9, T_j) \quad (465)$$

$$j = j \implies S_2^9|T_j \subseteq \text{LVis}(S_2^9, T_j) \quad (466)$$

$$T_j <_{S_2^9} T_k \implies S_2^9|T_k \not\subseteq \text{LVis}(S_2^9, T_j) \quad (467)$$

$$(465) \wedge (466) \wedge (467) \implies \text{LVis}(S_2^9, T_j) = S_2^9|T_i \cdot S_2^9|T_j \quad (468)$$

$$(468) \implies \text{LVis}(S_2^9, T_j)|x = [w_i(x)1 \rightarrow ok_i, r_j(x) \rightarrow 1, w_j(x)2 \rightarrow ok_j] \quad (469)$$

$$(479) \wedge (6) \implies \text{LVis}(S_2^9, T_j) \text{ is legal} \quad (470)$$

$$(470) \implies T_j \text{ in } S_2^9 \text{ is last-use legal in } S_2^9 \quad (471)$$

$$T_i <_{S_2^9} T_k \wedge (464) \implies S_2^9|T_i \subseteq \text{LVis}(S_2^9, T_k) \text{ or } S_2^9|T_i \not\subseteq \text{LVis}(S_2^9, T_k) \quad (472)$$

$$(472) \implies S_2^9|T_i \not\subseteq \text{LVis}(S_2^9, T_k) \quad (473)$$

$$w_j(x)2 \rightarrow ok_j \text{ is closing write on } x \text{ in } T_j \implies T_j \text{ is decided on } x \text{ in } S_2^9 \quad (474)$$

$$T_j <_{S_2^9} T_k \wedge (474) \implies S_2^9|T_j \subseteq \text{LVis}(S_2^9, T_k) \text{ or } S_2^9|T_j \not\subseteq \text{LVis}(S_2^9, T_k) \quad (475)$$

$$(475) \implies S_2^9|T_j \not\subseteq \text{LVis}(S_2^9, T_k) \quad (476)$$

$$k = k \implies S_2^9|T_k \subseteq \text{LVis}(S_2^9, T_k) \quad (477)$$

$$(473) \wedge (476) \wedge (477) \implies \text{LVis}(S_2^9, T_k) = S_2^9|T_k \quad (478)$$

$$(478) \implies \text{LVis}(S_2^9, T_k)|x = \emptyset \quad (479)$$

$$(479) \wedge (8) \implies \text{LVis}(S_2^9, T_k) \text{ is legal} \quad (480)$$

$$(480) \implies T_k \text{ in } S_2^9 \text{ is last-use legal in } S_2^9 \quad (481)$$

$$(453) \wedge (456) \wedge (463) \wedge (471) \wedge (481) \implies P_2^9 \text{ is final-state last-use opaque} \quad (482)$$

$$(483)$$

□

Let  $P_2^9$  be a prefix s.t.  $H_9 = P_2^9 \cdot [res_j(ok_j), r_k(x) \rightarrow 0]$ .

*Proof.*

$$\text{let } C_3^9 = \text{Compl}(P_3^9) = P_3^9 \cdot [\text{res}_j(A_j), \text{res}_k(A_k)] \quad (484)$$

$$\text{let } S_3^9 = C_3^9|T_i \cdot C_3^9|T_j \cdot C_3^9|T_k \quad (485)$$

$$S_3^9 \equiv C_3^9 \quad (486)$$

$$\text{real time order } <_{P_3^9} = \{T_i <_{P_3^9} T_k\} \quad (487)$$

$$\text{real time order } <_{S_3^9} = \{T_i <_{S_3^9} T_j, T_i <_{S_3^9} T_k, T_j <_{S_3^9} T_k\} \quad (488)$$

$$(487) \wedge (488) \implies <_{S_3^9} \subseteq <_{P_3^9} \quad (489)$$

$$i = i \implies S_3^9|T_i \subseteq \text{LVis}(S_3^9, T_i) \quad (490)$$

$$T_i <_{S_3^9} T_j \implies S_3^9|T_j \not\subseteq \text{LVis}(S_3^9, T_i) \quad (491)$$

$$T_i <_{S_3^9} T_k \implies S_3^9|T_k \not\subseteq \text{LVis}(S_3^9, T_i) \quad (492)$$

$$(490) \wedge (491) \wedge (492) \implies \text{LVis}(S_3^9, T_i) = S_3^9|T_i \quad (493)$$

$$(493) \implies \text{LVis}(S_3^9, T_i)|x = [w_i(x)1 \rightarrow ok_i] \quad (494)$$

$$(494) \wedge (2) \implies \text{LVis}(S_3^9, T_i) \text{ is legal} \quad (495)$$

$$(495) \implies T_i \text{ in } S_3^9 \text{ is last-use legal in } S_3^9 \quad (496)$$

$$w_i(x)1 \rightarrow ok_i \text{ is closing write on } x \text{ in } T_i \implies T_i \text{ is decided on } x \text{ in } S_3^9 \quad (497)$$

$$T_i <_{S_3^9} T_j \wedge (497) \implies S_3^9|T_i \subseteq \text{LVis}(S_3^9, T_j) \quad (498)$$

$$j = j \implies S_3^9|T_j \subseteq \text{LVis}(S_3^9, T_j) \quad (499)$$

$$T_j <_{S_3^9} T_k \implies S_3^9|T_k \not\subseteq \text{LVis}(S_3^9, T_j) \quad (500)$$

$$(498) \wedge (499) \wedge (500) \implies \text{LVis}(S_3^9, T_j) = S_3^9|T_i \cdot S_3^9|T_j \quad (501)$$

$$(501) \implies \text{LVis}(S_3^9, T_j)|x = [w_i(x)1 \rightarrow ok_i, r_j(x) \rightarrow 1, w_j(x)2 \rightarrow A_j] \quad (502)$$

$$(512) \wedge (7) \implies \text{LVis}(S_3^9, T_j) \text{ is legal} \quad (503)$$

$$(503) \implies T_j \text{ in } S_3^9 \text{ is last-use legal in } S_3^9 \quad (504)$$

$$T_i <_{S_3^9} T_k \wedge (497) \implies S_3^9|T_i \subseteq \text{LVis}(S_3^9, T_k) \text{ or } S_3^9|T_i \not\subseteq \text{LVis}(S_3^9, T_k) \quad (505)$$

$$(505) \implies S_3^9|T_i \not\subseteq \text{LVis}(S_3^9, T_k) \quad (506)$$

$$w_j(x)2 \rightarrow ok_j \text{ is closing write on } x \text{ in } T_j \implies T_j \text{ is decided on } x \text{ in } S_3^9 \quad (507)$$

$$T_j <_{S_3^9} T_k \wedge (507) \implies S_3^9|T_j \subseteq \text{LVis}(S_3^9, T_k) \text{ or } S_3^9|T_j \not\subseteq \text{LVis}(S_3^9, T_k) \quad (508)$$

$$(508) \implies S_3^9|T_j \not\subseteq \text{LVis}(S_3^9, T_k) \quad (509)$$

$$k = k \implies S_3^9|T_k \subseteq \text{LVis}(S_3^9, T_k) \quad (510)$$

$$(506) \wedge (509) \wedge (510) \implies \text{LVis}(S_3^9, T_k) = S_3^9|T_k \quad (511)$$

$$(511) \implies \text{LVis}(S_3^9, T_k)|x = \emptyset \quad (512)$$

$$(512) \wedge (8) \implies \text{LVis}(S_3^9, T_k) \text{ is legal} \quad (513)$$

$$(513) \implies T_k \text{ in } S_3^9 \text{ is last-use legal in } S_3^9 \quad (514)$$

$$(486) \wedge (489) \wedge (496) \wedge (504) \wedge (514) \implies P_3^9 \text{ is final-state last-use opaque} \quad (515)$$

□

Let  $P_4^9$  be a prefix s.t.  $H_9 = P_4^9 \cdot [w_j(x)2 \rightarrow ok_j, r_k(x) \rightarrow 0]$ .

**Lemma 52.**  $P_4^9$  is final-state last-use opaque.

*Proof.*

$$\text{let } C_4^9 = Compl(P_4^9) = P_4^9 \cdot [res_j(A_j), res_k(A_j)] \quad (516)$$

$$\text{let } S_4^9 = C_4^9|T_i \cdot C_4^9|T_j \cdot C_4^9|T_k \quad (517)$$

$$S_4^9 \equiv C_4^9 \quad (518)$$

$$\text{real time order } <_{P_4^9} = \{T_i <_{P_4^9} T_k\} \quad (519)$$

$$\text{real time order } <_{S_4^9} = \{T_i <_{S_4^9} T_j, T_i <_{S_4^9} T_k, T_j <_{S_4^9} T_k\} \quad (520)$$

$$(520) \wedge (520) \implies <_{S_4^9} \subseteq <_{P_4^9} \quad (521)$$

$$i = i \implies S_4^9|T_i \subseteq LVis(S_4^9, T_i) \quad (522)$$

$$T_i <_{S_4^9} T_j \implies S_4^9|T_j \not\subseteq LVis(S_4^9, T_i) \quad (523)$$

$$T_i <_{S_4^9} T_k \implies S_4^9|T_k \not\subseteq LVis(S_4^9, T_i) \quad (524)$$

$$(522) \wedge (523) \wedge (524) \implies LVis(S_4^9, T_i) = S_4^9|T_i \quad (525)$$

$$(525) \implies LVis(S_4^9, T_i)|x = [w_i(x)1 \rightarrow ok_i] \quad (526)$$

$$(526) \wedge (2) \implies LVis(S_4^9, T_i) \text{ is legal} \quad (527)$$

$$(527) \implies T_i \text{ in } S_4^9 \text{ is last-use legal in } S_4^9 \quad (528)$$

$$w_i(x)1 \rightarrow ok_i \text{ is closing write on } x \text{ in } T_i \implies T_i \text{ is decided on } x \text{ in } S_4^9 \quad (529)$$

$$T_i <_{S_4^9} T_j \wedge (529) \implies S_4^9|T_i \subseteq LVis(S_4^9, T_j) \quad (530)$$

$$j = j \implies S_4^9|T_j \subseteq LVis(S_4^9, T_j) \quad (531)$$

$$T_j <_{S_4^9} T_k \implies S_4^9|T_k \not\subseteq LVis(S_4^9, T_j) \quad (532)$$

$$(530) \wedge (531) \wedge (532) \implies LVis(S_4^9, T_j) = S_4^9|T_i \cdot S_4^9|T_j \quad (533)$$

$$(533) \implies LVis(S_4^9, T_j)|x = [w_i(x)1 \rightarrow ok_i, r_j(x) \rightarrow 1] \quad (534)$$

$$(545) \wedge (4) \implies LVis(S_4^9, T_j) \text{ is legal} \quad (535)$$

$$(535) \implies T_j \text{ in } S_4^9 \text{ is last-use legal in } S_4^9 \quad (536)$$

$$(537)$$

$$T_i <_{S_4^9} T_k \wedge (529) \implies S_4^9|T_i \subseteq LVis(S_4^9, T_k) \text{ or } S_4^9|T_i \not\subseteq LVis(S_4^9, T_k) \quad (538)$$

$$(538) \implies S_4^9|T_i \not\subseteq LVis(S_4^9, T_k) \quad (539)$$

$$w_j(x)2 \rightarrow ok_j \text{ is closing write on } x \text{ in } T_j \implies T_j \text{ is decided on } x \text{ in } S_4^9 \quad (540)$$

$$T_j <_{S_4^9} T_k \wedge (540) \implies S_4^9|T_j \subseteq LVis(S_4^9, T_k) \text{ or } S_4^9|T_j \not\subseteq LVis(S_4^9, T_k) \quad (541)$$

$$(541) \implies S_4^9|T_j \not\subseteq LVis(S_4^9, T_k) \quad (542)$$

$$k = k \implies S_4^9|T_k \subseteq LVis(S_4^9, T_k) \quad (543)$$

$$(539) \wedge (542) \wedge (543) \implies LVis(S_4^9, T_k) = S_4^9|T_k \quad (544)$$

$$(544) \implies LVis(S_4^9, T_k)|x = \emptyset \quad (545)$$

$$(545) \wedge (8) \implies LVis(S_4^9, T_k) \text{ is legal} \quad (546)$$

$$(546) \implies T_k \text{ in } S_4^9 \text{ is last-use legal in } S_4^9 \quad (547)$$

$$(518) \wedge (521) \wedge (528) \wedge (536) \wedge (547) \implies P_4^9 \text{ is final-state last-use opaque} \quad (549)$$

□

Let  $P_5^9$  be a prefix s.t.  $H_9 = P_5^9 \cdot [start_k, w_j(x)2 \rightarrow ok_j, r_k(x) \rightarrow 0]$ .

**Lemma 53.**  $P_4^9$  is final-state last-use opaque.

*Proof.*

$$P_5^9 = P_2^3 \wedge \text{Lemma 28} \quad (550)$$

□

Let  $P_p^9$  be any prefix of  $P_5^9$ .

**Lemma 54.**  $P_p^9$  is final-state last-use opaque.

*Proof.*

$$H_1 = P_4^1 \cdot R \wedge \text{Corollary 9} \implies P_4^1 \text{ is last-use plague} \quad (551)$$

$$P_4^1 = P_3^3 \wedge (551) \implies P_p^9 \text{ is last-use lopaque} \quad (552)$$

$$(552) \implies P_p^9 \text{ is final-state last-use lopaque} \quad (553)$$

□

**Lemma 55.**  $H_9$  is last-use opaque.

*Proof.* Since, from Lemmas 51–54, all prefixes of  $H_9$  are final-state last-use opaque, then by Def. 22  $H_9$  is last-use opaque. □

**Corollary 10.** Any prefix of  $H_9$  is last-use opaque.

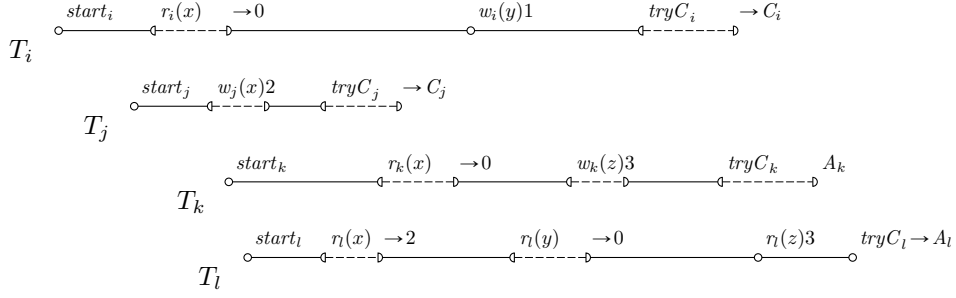


Figure 37: TMS1 history example [11].

## B Property Comparison

VWC is incomparable to last-use opacity.

**Lemma 56.** *There exists a last-use-opaque history  $H$  that is not virtual world consistent.*

*sketch.* Since, as an extension of by Lemma 21, last-use opacity supports early release, then by Def. 3 and (by Def. 5) from Lemma 55 there exists some last-use-opaque history where some transaction reads from a live transaction and aborts. Since, by Theorem 8 VWC, does not support aborting releasing transactions, then, by the same definitions, such a history is not VWC. Hence a history with a transaction releasing early may be last-use-opaque but not VWC.  $\square$

**Theorem 17.** *There exists a virtual world consistent history  $H$  that is not last-use-opaque.*

*sketch.* Since each transaction in a VWC history can be explained by a different causal past from other transactions, it is possible that in a correct VWC history transactions do not agree on the order of operations in the sequential witness history. However, in order for  $H$  to be last-use-opaque the legality of transactions needs to be established using a single sequential history with a single order of operations. Thus, it is possible for a VWC history not to be last-use-opaque.  $\square$

Since TMS1 is incomparable to last-use opacity.

**Theorem 18.** *There exists a last-use-opaque history  $H$  that is not TMS1.*

*sketch.* Since, as an extension of by Lemma 21, last-use opacity supports early release, then by Def. 3 and 2 there exist histories that are last-use-opaque where some transaction reads from a live transaction. Since, by Theorem 5 TMS1, does not support early release, then, by the same definitions, histories containing early release are not TMS1. Hence a history with a transaction releasing early may be last-use-opaque but not TMS1.  $\square$

**Theorem 19.** *There exists a TMS1 history  $H$  that is not last-use-opaque.*

*sketch.* Let history  $H$  be the history presented in Fig. 37. In [11] (Fig. 6 therein) the authors show that the history satisfies TMS1. The same history is not last-use opaque. Note that if  $Vis(S, T_i)$  is to be legal, in any  $S$  equivalent to  $H$ ,  $T_i <_S T_j$ , because  $T_i$  reads 0 from  $x$

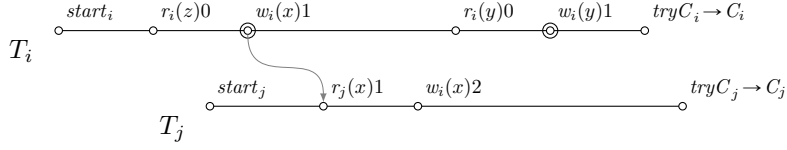


Figure 38: Last-use-opaque history that does not satisfy elastic opacity.

and  $T_j$  writes 2 to  $x$  (and commits). In addition,  $T_j <_S T_l$ , because  $T_l$  reads 2 from  $x$  and  $T_k <_S T_l$ , because  $T_l$  reads  $z$  from  $T_k$ . Then, by extension  $T_i <_S T_j <_S T_l$ . However, note that in any  $S$  it must be that  $T_l <_S T_i$ , because  $T_l$  reads  $y$  from  $T_i$ , which is a contradiction. Thus,  $H$  is not last-use opaque.  $\square$

TMS2 is strictly stronger than last-use opacity.

**Proposition 1.** *All TMS2 histories are last-use-opaque.*

*sketch.* The authors of [11] believe (but do not demonstrate) that all opaque histories satisfy TMS2. If this is the case, then, since all opaque histories are last-use-opaque (Lemma 9), then it is true that all last-use-opaque histories satisfy TMS2. Thus, we believe the proposition is true, pending a demonstration that all opaque histories satisfy TMS2.  $\square$

Last-use Opacity and elastic opacity are incomparable.

**Lemma 57.** *There exists an elastic opaque history  $H$  that is not last-use-opaque.*

*sketch.* Since elastic opaque histories may not be serializable [13], and since, as all last-use-opaque histories trivially require serializability then some elastic opaque histories are not last-use-opaque.  $\square$

**Lemma 58.** *There exists a last-use-opaque history  $H$  that is not elastic opaque.*

*sketch.* Let history  $H$  be the history presented in Fig. 38. It should be straightforward to see that  $H$  is last-use-opaque for an equivalent sequential history  $S = H|T_i \cdot H|T_j$ . Operations on  $z$  are always justified in any sequential equivalent history since they are all within  $T_i$  and their effects are not visible in  $T_j$ . The read operation on  $y$  is expected to read 0 since it is not preceded in  $S$  by any write, and it does read 0. Thus operations on  $y$  and  $z$  will not break legality of either  $T_i$  or  $T_j$ . With that in mind, the history can be shown to be last-use opaque by analogy to Lemma 21.

On the other hand, let  $T_i$  be an elastic transaction. The only possible well-formed cut of  $H|T_i$  is  $C_i = \{[r(z)0, w(x)1, r(y)0, w(y)1]\}$ . (In particular, the following cut is not well-formed, since  $w(x)1$  and  $w(y)0$  are in two different subhistories of the cut:  $C'_i = \{[r(z)0, w(x)1], [r(y)0, w(y)1]\}$ ). Let  $f_C(H)$  be a cutting function that applies cut  $C$ . Then, since the cut contains only one subhistory, it should be straightforward to see that  $f_C(H) = H$ . Then, we note that  $H$  contains an operation in  $H|T_j$  that reads the value of  $x$  from  $H|T_i$  and  $T_i$  is live. That means that in the prefix  $P$  of  $H$  s.t.  $H = P \cdot [try C_i \rightarrow C_i, try C_j \rightarrow C_j]$  both transactions will be aborted in any completion of  $P$ , so for any sequential equivalent history  $S$   $Vis(S, T_i)$  will not contain  $S|T_j$ , since either  $T_j$  is aborted in any  $S$ . Therefore  $Vis(S, T_i)$  will not justify reading 1 from  $x$  and will not be legal, causing  $P$  not to be final state opaque (Def. 7), which in turn means that  $H$  is not opaque (Def. 8).  $\square$

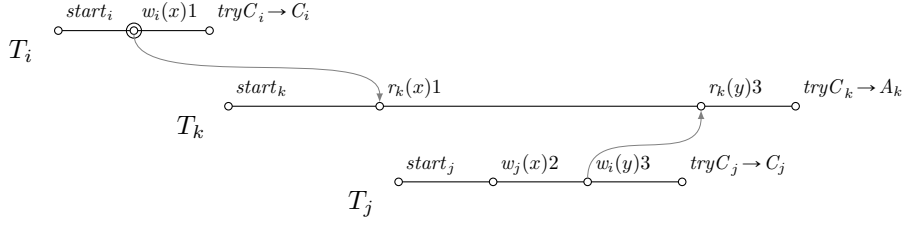


Figure 39: ACA history that does not satisfy elastic last-use opacity.

Last-use opacity is strictly stronger than recoverability.

**Lemma 59.** *Any last-use-opaque history  $H$  is recoverable.*

*sketch.* Let us assume that  $H$  is not recoverable. Then there must be some transactions  $T_i$  and  $T_j$  s.t.  $T_j$  reads from  $T_i$  and then  $T_j$  commits before  $T_i$ . By analogy to Lemma 25, such a history will contain a prefix  $P$  where any completion will contain an aborted  $T_i$  and a committed  $T_j$ , so for any equivalent sequential history  $S$   $Vis(S, T_j)$  will not contain  $S|T_i$ . Since  $T_i$  reads from  $T_i$  then such  $Vis(S, T_j)$  will not be legal, so by Def. 21  $P$  is not last-use opaque and thus, by Def. 22,  $H$  is not last-use opaque, which is a contradiction.  $\square$

ACA is incomparable to last-use opacity.

**Lemma 60.** *There exists a last-use-opaque history  $H$  that does not avoid cascading aborts.*

*sketch.* Lemma 21 demonstrates that  $H_1$  is last-use-opaque. However, since  $T_i$  reads from  $T_j$  in  $H_1$  and  $r_j(x) \rightarrow v <_{H_1} tryC_i \rightarrow C_i$  the history is not ACA, since it contradicts Def. 15.  $\square$

**Lemma 61.** *There exists an ACA history  $H$  that is not last-use-opaque.*

*sketch.* The history in Fig. 39 is shown to be ACA in [4]. However, note, that  $Compl(H) = H$ , and given any sequential  $S \equiv Compl(H)$   $T_k$   $T_k$  must follow both  $T_i$  and  $T_k$  in  $S$  because  $T_k$  reads from both transactions. Since  $T_i <_H^{rt} T_j$  and  $T_i <_H^{rt} T_k$ , then  $T_i$  must precede both other transactions in  $S$ . Hence,  $S = H|T_i \cdot H|T_j \cdot H|T_k$ , so  $Vis(S, T_k) = S$  and therefore  $Vis(S, T_k)$  is illegal because  $r_k(x) \rightarrow 1$  is preceded in  $Vis(S, T_k)|x$  by  $r_k(x) \rightarrow 2$ .  $\square$

Strictness and last-use opacity are also incomparable.

**Theorem 20.** *There exists a last-use-opaque history  $H$  that is not strict.*

*sketch.* Since any strict history is also ACA [4], and since Lemma 60 shows that not all last-use-opaque histories are ACA, then not all last-use-opaque histories are strict.  $\square$

**Theorem 21.** *There exists a strict history  $H$  that is not last-use-opaque.*

*sketch.* The history in Fig. 39 is shown to be strict in [4]. However, as we show in Lemma 61, this history is not last-use-opaque.  $\square$

Rigorousness is strictly stronger than last-use opacity.

**Lemma 62.** *Any rigorous history  $H$  is last-use-opaque.*

*sketch.* Since [4] demonstrates that rigorous histories are opaque, and since we show in Lemma 9 that opaque histories are also last-use-opaque, then all rigorous histories are last-use-opaque.  $\square$

Live opacity is stronger than last-use opacity.

**Lemma 63.** *Any live opaque history  $H$  is last-use-opaque.*

*sketch.* Since  $H$  is live opaque there exists a sequential history  $S$  that justifies serializability of  $H$  and an extension  $S'$  of  $S$  where if transaction  $T_i$  is not in  $S$  then it is replaced in  $S'$  by  $T_i^{gr}$  containing only non-local reads.  $S'$  is legal and preserves the real-time order of  $H$  (accounting for replaced transactions). In addition, from Theorem 11, no transaction in  $H$  reads from a live transaction (in any prefix of  $H$ ). Therefore, since  $S'$  is legal, any read operation  $op_i = r_i(x) \rightarrow v$  in  $H$  that is preceded  $w_j(x)v \rightarrow u$  in  $H$ ,  $T_j$  is committed in  $S$  and is included in  $S'$  in full.

Let  $S''$  be a sequential history constructed by replacing the operations removed to create  $S'$  where if  $T_i \in H$  and  $T_i \not\in S$  then  $T_i$  is aborted in  $S''$ .  $S''$  preserves the real time order of  $H$  and  $S'' \equiv H$ . Note that, since  $S'$  is legal, if some write  $op^w$  is in  $S''$  and not in  $S'$ , then there is no non-local read operation  $op^r$  reading the value written by  $op^w$ . Hence any operation reading the value written by  $op^w$  is local, and since all local reads in transactions that are replaced in  $S'$  read legal values (by Def. 12), then all reads reading from any  $op_w$  read legal values in  $S''$ . Since  $S'$  is legal, then all reads reading from transactions that are in  $S$  read legal values in  $S'$ . Since  $S'' \equiv H$ , then these read and write operations also read legal values in  $S''$ . Because of this, and since no transaction reads from another live transaction,  $Vis(S'', T_i)$  will be legal for any transaction in  $S''$ . In addition,  $LVis(S'', T_i)$  will be legal for any aborted transaction in  $S''$ . Therefore any live opaque  $H$  will be final state last-use opaque. Since any prefix of  $H$  is also live opaque, then any prefix will also be final-state last-use opaque, hence  $H$  is opaque.  $\square$

## C $\beta$ -last-use opacity

**Definition 26** ( $\beta$ -Closing Write Invocation). *Given a program  $\mathbb{P}$ , a set of processes  $\Pi$  executing  $\mathbb{P}$  and a history  $H$  s.t.  $H \models \mathcal{E}(\mathbb{P}, \Pi)$ , i.e.  $H \in \mathbb{H}^{\mathbb{P}, \Pi}$ , an invocation  $inv_i(w(x)v)$  is the closing write invocation on some variable  $x$  by transaction  $T_i$  in  $H$ , if for any history  $H' \in \mathbb{H}^{\mathbb{P}, \Pi}$  for which  $H$  is a prefix (i.e.,  $H' = H \cdot R$ ) there is no operation invocation  $inv'$  s.t.  $inv_i(w(x)v)$  precedes  $inv'$  in  $H'|T_i$  where (a)  $inv' = inv_i(w(x)u)$  or (b)  $inv' = inv_i(tryA)$ .*

**Definition 27** ( $\beta$ -Closing Write). *Given program  $\mathbb{P}$ , a set of processes  $\Pi$  executing  $\mathbb{P}$  and history  $H$  s.t.  $H \models \mathcal{E}(\mathbb{P}, \Pi)$ , an operation execution is the closing write on some variable  $x$  by transaction  $T_i$  in  $H$  if it comprises of an invocation and a response other than  $A_i$ , and the invocation is the  $\beta$ -closing write invocation on  $x$  by  $T_i$  in  $H$ .*

**Definition 28** (Transaction  $\beta$ -Decided on  $x$ ). *Given a program  $\mathbb{P}$ , a set of processes  $\Pi$  and a history  $H$  s.t.  $H \models \mathcal{E}(\mathbb{P}, \Pi)$ , we say transaction  $T_i \in H$   $\beta$ -decided on variable  $x$  in  $H$  iff  $H|T_i$  contains a complete write operation execution  $w_i(x)v \rightarrow ok_i$  that is the  $\beta$ -closing write on  $x$ .*

Given some history  $H$ , let  $\hat{\mathbb{T}}_\beta^H$  be a set of transactions s.t.  $T_i \in \hat{\mathbb{T}}_\beta^H$  iff there is some variable  $x$  s.t.  $T_i$   $\beta$ -decided on  $x$  in  $H$ .

Given any  $T_i \in H$ ,  $H|^\beta T_i$  is the longest subsequence of  $H|T_i$  s.t.:

- a)  $H|^\beta T_i$  contains  $start_i \rightarrow u$ , and
- b) for any variable  $x$ , if  $T_i$   $\beta$ -decided on  $x$  in  $H$ , then  $H|^\beta T_i$  contains  $(H|T_i)|x$ .

In addition,  $H|^\beta T_i$  is a sequence s.t.  $H|^\beta T_i = H|^\beta T_i \cdot [try C_i \rightarrow C_i]$ .

Given a sequential history  $S$  s.t.  $S \equiv H$ ,  $\beta LVis(S, T_i)$  is the longest subhistory of  $S$ , s.t. for each  $T_j \in S$ :

- a)  $S|T_j \subseteq \beta LVis(S, T_i)$  if  $i = j$  or  $T_j$  is committed in  $S$ , or
- b)  $S|^\beta T_j \subseteq \beta LVis(S, T_i)$  if  $T_j$  is not committed in  $S$  but  $T_j \in \hat{\mathbb{T}}_\beta^H$ .

Given a sequential history  $S$  and a transaction  $T_i \in S$ , we then say that transaction  $T_i$  is  $\beta$ -last-use legal in  $S$  if  $\beta LVis(S, T_i)$  is legal.

**Definition 29** (Final-state  $\beta$ -Last-use Opacity). *A finite history  $H$  is final-state  $\beta$ -last-use opaque if, and only if, there exists a sequential history  $S$  equivalent to any completion of  $H$  s.t.,*

- a)  $S$  preserves the real-time order of  $H$ ,
- b) every transaction in  $S$  that is committed in  $S$  is legal in  $S$ ,
- c) every transaction in  $S$  that is not committed in  $S$  is  $\beta$ -last-use legal in  $S$ .

**Definition 30** ( $\beta$ -Last-use Opacity). *A history  $H$  is  $\beta$ -last-use opaque if, and only if, every finite prefix of  $H$  is final-state  $\beta$ -last-use opaque.*

## D SVA Correctness Proof

Since the values used within writes are under the control of the program (rather than SVA) we simply assume that they are within the domain of the variables.

**Assumption 1** (Values Within Domain). *For any transaction  $T_i$  in any SVA history  $H$  given variable  $x$  with the domain of  $\mathbb{D}$ , if  $w_i(x)v \rightarrow u \in H|T_i$ , then  $v \in \mathbb{D}$ .*

**Definition 31** (Direct Precedence). *For operations  $op_i, op_j \in H$ ,  $op_j \ll_H op_i$  iff  $op_j <_H op_i$  and  $\nexists op_k \in H$  s.t.  $op_j <_H op_k <_H op_i$ .*

**Definition 32** (Operation Execution Conditional). *Given predicate  $P$  and operation  $op$   $P \rightarrow op$  denotes that  $P$  is true only if  $op$  executes.*

**Definition 33** (Operation Execution Converse). *Given predicate  $P$  and operation  $op$   $P \leftarrow op$  denotes that  $op$  executes only if  $P$  is true.*

Let there be any  $\mathbb{P}, \Pi, H \models \mathcal{E}(\mathbb{P}, \Pi)$ ,  $op_i \in H|T_i$ .

**Definition 34.**  $op_i$  is closing access on  $x$  in  $T_i$ , denoted  $op_i = \ddot{op}_i^x$  if both:

- a)  $op_i$  is closing read on  $x$  in  $T_i$  or  $op_i$  is closing write on  $x$  in  $T_i$ , and  
b)  $\nexists op'_i \in H$  s.t.  $op_i <_H op'_i$  and  $op'_i$  is closing read on  $x$  in  $T_i$  or  $op'_i$  is closing write on  $x$  in  $T_i$ .

Let there be any  $\mathbb{P}, \Pi, H \models \mathcal{E}(\mathbb{P}, \Pi)$ ,  $op_i \in H|T_i$ ,  $op_i = \begin{cases} r_i(x) \rightarrow v, \\ w_i(x)v \rightarrow ok_i. \end{cases}$

**Lemma 64** (Access Condition).  $lv(x) = pv_i(x) - 1 \leftarrow op_i$ .

*Proof.* Condition at line 13 dominates access at line 17.  $\square$

**Lemma 65** (Abort Condition).  $ltv(x) = pv_i(x) - 1 \leftarrow res_i(A_i)$ .

*Proof.* Access condition at line 27 dominates dismiss at line 28 in procedure abort for each variable. Hence, all variables must pass line 27 before abort concludes.  $\square$

**Lemma 66** (Commit Condition).  $ltv(x) = pv_i(x) - 1 \leftarrow res_i(C_i)$ .

*Proof.* By analogy to Lemma 65.  $\square$

**Lemma 67** (Early Release). If  $op_i = \ddot{op}_i^x$  then  $lv(x) = pv_i(x) \rightarrow op_i$ .

*Proof.*  $lv(x)$  can be set by  $T_i$  at line 21 and at line 58. The former is set during the last access on some  $x$  in  $T_j$  (line 19 dominates line 21). The latter is set during commit, which means that if any closing access was present, it was executed prior to commit.  $\square$

Let  $r_i = \begin{cases} res_i(A_i), \\ res_i(C_i). \end{cases}$

**Lemma 68** (Release). If  $\nexists op'_i \in H|T_i$  s.t.  $op'_i = \ddot{op}_i^x$  and  $x \in ASet(T_i)$  then  $lv(x) = pv_i(x) \rightarrow r_i$ .

*Proof.* If  $op'_i$  is not closing access then line 19 will not be passed, so only assignment of  $lv(x)$  is in line 58 which execute only during commit or abort.  $\square$

**Lemma 69** (Terminal Release). If  $x \in ASet(T_i)$  then  $lv(x) = pv_i(x) \rightarrow r_i$ .

*Proof.*  $ltv(x)$  can be set only in line 31 or line 43, which are part of abort and commit, respectively.  $\square$

Let there be any  $H, T_i \in H, T_j \in H, op_i \in H|T_i, op_i = r_i(x) \rightarrow u, op_j \in H|T_j, op_j = w_j(x)v \rightarrow ok_j$ .

**Lemma 70** (No Buffering). If  $op_j \ll_{H|x} op_i$  and not  $op_j <_H res_j(A_j) <_H op_i$  then  $u = v$ .

**Lemma 71** (Revert On Abort). If  $op_j \ll_{H|x} op_i$  and  $op_j <_H res_j(A_j) <_H op_i$  then  $u \neq v$ .

*Proof.* If abort is executed then the restore procedure is executed for all  $x \in ASet(T_i)$ . Thus, line 63 restores  $x$  to value  $v'$  which is acquired before the any operation on  $x$  is executed by  $T_i$ , hence  $v' \neq v$ , so  $u \neq v$ .  $\square$

Let  $H|start$  be a subhistory of  $H$  that for each  $T_j \in H$  contains only the operation  $start_j$ .

**Lemma 72** (Consecutive Versions). *If  $x \in ASet(T_i) \cap ASet(T_j)$  and  $inv_i(start_i) \ll_{H|start} inv_i(start_j)$  then  $pv_i(x) - 1 = pv_j(x)$ .*

*Proof.* If  $T_i$  returns at line 3 for  $x$  then no  $T_j$  s.t.  $x \in ASet(T_j)$  returns at line 3 until  $T_i$  executes line 9 for  $x$ . Hence,  $T_i$  alone increments  $gv(x)$  at line 5 and sets  $pv_i(x)$  to the new value of  $gv(x)$ . If  $start_i \ll_{H|start} start_j$  then  $T_i$  will return at line 3 and  $T_j$  will return next. No other transaction will return at line 3 between  $T_i$  and  $T_j$ .  $\square$

**Lemma 73** (Unique Versions). *If  $x \in ASet(T_i) \cap ASet(T_j)$  then  $pv_i(x) \neq pv_j(x)$ .*

*Proof.* From Lemma 72.  $\square$

**Lemma 74** (Monotonic Versions). *If  $x \in ASet(T_i) \cap ASet(T_j)$  and  $inv_i(start_i) <_{H|start} inv_i(start_j)$  then  $pv_i(x) < pv_j(x)$ .*

*Proof.* From Lemma 72, Lemma 73.  $\square$

**Definition 35** (Version Order). *Let  $<_x$  be an order s.t.  $T_i <_x T_j$  iff  $pv_i(x) < pv_j(x)$ .*

**Lemma 75** (Forced Abort Condition).  $rv_i(x) < cv(x) \rightarrow res_i(A_i)$ .

*Proof.* Condition at line 15 dominates abort at line 16. Condition at line 39 dominates abort at line 40.  $\square$

Let there be any  $\mathbb{P}, \Pi, H \models \mathcal{E}(\mathbb{P}, \Pi)$ ,  $op_i \in H|T_i$ ,  $op_i = \begin{cases} r_i(x) \rightarrow v, \\ w_i(x)v \rightarrow ok_i. \end{cases}$

**Lemma 76.**  $cv(x) < rv_i(x) \leftarrow op_i$ .

*Proof.* Condition at line 15 dominates abort at line 16.  $\square$

**Lemma 77** (Current Version Early Release). *If  $op_i = \ddot{op}_i^x$  then  $cv(x) = rv_i(x) \rightarrow op_i$ .*

*Proof.* By analogy to Lemma 67.  $\square$

**Lemma 78** (Current Version Release). *If  $\nexists op_i \in H|T_i$  s.t.  $op_i = \ddot{op}_i^x$  and  $x \in ASet(T_i)$  then  $cv(x) = rv_i(x) \rightarrow r_i$ .*

*Proof.* By analogy to Lemma 68.  $\square$

**Lemma 79.**  $cv(x) = rv_i(x) \leftarrow res_i(A_i)$ .

*Proof.* From Lemma 68 and Lemma 78.  $\square$

Let there be any  $\mathbb{P}, \Pi, H \models \mathcal{E}(\mathbb{P}, \Pi)$ ,  $T_i \in H$ ,  $T_j \in H$ ,  $op_i \in H|T_i$ ,  $op_j \in H|T_j$ ,  $op_i = \begin{cases} r_i(x) \rightarrow v, \\ w_i(x)v \rightarrow ok_i, \end{cases}$   $op_j = \begin{cases} r_j(x) \rightarrow v, \\ w_j(x)v \rightarrow ok_j. \end{cases}$

**Lemma 80** (Access Order).  $pv_i(x) < pv_j(x) \Leftrightarrow op_i <_H op_j$ .

*Proof.* From Lemma 64 and Lemma 74.  $\square$

Let there be any  $H$ ,  $T_i \in H$ ,  $T_j \in H$ ,  $op_i \in H|T_i$ ,  $op_i = r_i(x) \rightarrow u$ ,  $op_j \in H|T_j$ ,  $op_j = w_j(x)v \rightarrow ok_j$ .

Let there be any  $\mathbb{P}, \Pi, H \models \mathcal{E}(\mathbb{P}, \Pi)$ .

**Lemma 81** (Access Prefix). *If  $lv(x) = pv_j(x)$  then  $\forall T_k \in H$  s.t.  $pv_k(x) < pv_i(x)$  either  $res_k(C_k) \in H|T_k$ ,  $res_k(A_k) \in H|T_k$ , or  $\ddot{op}_k^x \in H|T_k$ .*

*Proof.*

$$\forall T_l, T_k \in H \text{ s.t. } pv_l(x) = pv_k(x) - 1 : \quad (554)$$

$$\text{Lemma 64} \implies lv(x) = pv_k(x) - 1 \leftarrow op_k \quad (555)$$

$$(554) \wedge (555) \implies lv(x) = pv_l(x) \leftarrow op_k \quad (556)$$

$$\text{Lemma 67} \implies lv(x) = pv_l(x) \rightarrow \ddot{op}_l^x \quad (557)$$

$$\text{Lemma 68} \implies lv(x) = pv_l(x) \rightarrow r \text{ where } r = res_i(A_i) \text{ or } r = res_i(C_i) \quad (558)$$

$$(557) \wedge (558) \implies T_l \text{ is committed, aborted or decided on } x \quad (559)$$

Trivially extends for any  $T_l, T_k$  s.t.  $pv_l(x) < pv_k(x)$ .  $\square$

**Lemma 82** (C). *If  $ltv(x) = pv_j(x)$  then  $\forall T_k \in H$  s.t.  $pv_k(x) < pv_i(x)$  either  $res_k(C_k) \in H|T_k$ , or  $res_k(A_k) \in H|T_k$ .*

*Proof.*

$$\forall T_l, T_k \in H \text{ s.t. } pv_l(x) = pv_k(x) - 1 : \quad (560)$$

$$\text{Lemma 66} \implies ltv(x) = pv_k(x) - 1 \leftarrow res_k(C_i) \quad (561)$$

$$(560) \wedge (561) \implies ltv(x) = pv_l(x) \leftarrow op_k \quad (562)$$

$$\text{Lemma 69} \implies ltv(x) = pv_l(x) \rightarrow r \text{ where } r = res_i(A_i) \text{ or } r = res_i(C_i) \quad (563)$$

$$(563) \implies T_l \text{ is committed or aborted} \quad (564)$$

Trivially extends for any  $T_l, T_k$  s.t.  $pv_l(x) < pv_k(x)$ .  $\square$

Let there be any  $H, T_i \in H, T_j \in H, op_i \in H|T_i, op_i = r_i(x) \rightarrow u, op_j \in H|T_j, op_j = w_j(x)v \rightarrow ok_j$ .

**Lemma 83** (Forced Abort). *If  $x \in ASet(T_i) \cap ASet(T_j)$  and  $res_j(A_j) \in H|T_j$  and  $op_i <_H res_j(A_j)$  then  $res_i(A_i) \in H|T_i$ .*

*Proof.*

$$res_j(A_j) \in H|T_j \wedge \text{Lemma 79} \implies cv(x) = pv_jx \leftarrow res_j(A_j) \quad (565)$$

$$\text{Lemma 74} \implies pv_j(x) < pv_i(x) \quad (566)$$

$$(565) \wedge (566) \implies cv(x) < pvix \leftarrow res_j(A_j) \quad (567)$$

$$\text{Lemma 76} \implies cv(x) = rv_i(x) \leftarrow op_i \implies rv_i(x) = pv_j(x) \quad (568)$$

$$(568) \wedge (566) \implies rv_i(x) > pv_i(x) \quad (569)$$

$$(569) \wedge (567) \implies rv_i(x) < cv(x) \quad (570)$$

$$(570) \implies rv_i(x) < cv(x) \rightarrow res_i(A_i) \implies res_i(A_i) \in H|T_i \quad (571)$$

$\square$

Let there be any  $\mathbb{P}, \Pi, H \models \mathcal{E}(\mathbb{P}, \Pi), T_i \in H, T_j \in H, op_i \in H|T_i, op_j \in H|T_j, op_i = \begin{cases} r_i(x) \rightarrow v, \\ w_i(x)v \rightarrow ok_i, \end{cases} op_j = \begin{cases} r_j(x) \rightarrow v, \\ w_j(x)v \rightarrow ok_j. \end{cases}$

**Definition 36** (Completion Construction).  $H_C = \text{Compl}(H)$  s.t.  $\forall T_k \in H, \text{res}_k(C_k) \notin H|T_k \Leftrightarrow \text{res}_k(A_k) \in H_C|T_k$

**Definition 37** (Sequential History Construction).  $\hat{S}_H$  is a sequential history s.t.  $\hat{S}_H \equiv H_C$  and  $T_i <_{H_C} T_j \Rightarrow T_i <_{\hat{S}_H} T_j$  and  $T_i <_x T_j \Rightarrow T_i <_{\hat{S}_H} T_j$ .

Let there be any  $H, T_i \in H, T_j \in H, op_i \in H|T_i, op_i = r_i(x) \rightarrow u, op_j \in H|T_j, op_j = w_j(x)v \rightarrow ok_j$ .

**Lemma 84.** If  $T_i$  reads  $x$  from  $T_j$  then  $T_j$  is committed in  $H$  or  $T_j$  is decided on  $x$  in  $H$ .

*Proof.*

$$T_i \text{ reads } x \text{ from } T_j \Rightarrow op_i = r_i(x) \rightarrow v \wedge op_j = w_j(x)v \rightarrow ok_i \wedge op_j <_H op_i \quad (572)$$

$$\text{Lemma 64} \Rightarrow 1v(x) = \text{pv}_x(-)1 \leftarrow op_i \quad (573)$$

$$\text{Lemma 80} \wedge op_j <_H op_i \Rightarrow \text{pv}_j(x) < \text{pv}_i(x) \quad (574)$$

$$(574) \wedge \text{Lemma 81} \Rightarrow T_j \text{ is committed, aborted, or decided on } x \quad (575)$$

Let us assume that  $T_j$  is aborted:

$$op_i < \text{res}_j(A_j) : \text{Lemma 72} \Rightarrow v \neq v \Rightarrow \text{contradiction} \quad (576)$$

$$\text{res}_j(A_j) <_H op_i : \text{Lemma 67} \Rightarrow 1v(x) = \text{pv}_x(\rightarrow)op_i \wedge op_i = \ddot{op}_i^x \quad (577)$$

Thus,  $T_i$  is committed or decided on  $x$ .  $\square$

**Corollary 11.** If  $P$  is any prefix of  $H$ , then if  $T_i$  reads  $x$  from  $T_j$  in  $P$  then  $T_j$  is committed in  $P$  or  $T_j$  is decided on  $x$  in  $P$ .

**Lemma 85.** If  $T_i$  reads  $x$  from  $T_j$  and  $T_j$  is committed in  $H$  then  $T_j$  is committed in  $H$ .

*Proof.*

$$T_i \text{ reads } x \text{ from } T_j \Rightarrow op_i = r_i(x) \rightarrow v \wedge op_j = w_j(x)v \rightarrow ok_i \wedge op_j <_H op_i \quad (578)$$

$$\text{Lemma 66} \Rightarrow 1\text{tv}(x) = \text{pv}_k(x) - 1 \leftarrow \text{res}_i(ok_i) \quad (579)$$

$$\text{Lemma 80} \wedge op_j <_H op_i \Rightarrow \text{pv}_j(x) < \text{pv}_i(x) \quad (580)$$

$$\text{Lemma 82} \wedge (579) \wedge (580) \Rightarrow r \in H|T_j \text{ where } r = \text{res}_j(A_j) \text{ or } r = \text{res}_j(C_j) \quad (581)$$

$$\text{Lemma 83} \Rightarrow \text{if } \text{res}_j(A_j) \in H|T_j \text{ then } \text{res}_j(A_j) \in H|T_j \Rightarrow \text{contradiction} \quad (582)$$

$$(582) \Rightarrow \text{res}_i(A_i) \in H|T_i \quad (583)$$

$\square$

Let there be any  $\mathbb{P}, \Pi, H \models \mathcal{E}(\mathbb{P}, \Pi), T_i \in H, op_i = r_i(x) \rightarrow v, op_i \in H|T_i$ .

**Lemma 86.** If  $\text{res}_i(C_i) \in \hat{S}_H|T_i$  then  $\exists op_j = w_j(x)v \rightarrow ok_j \in \text{Vis}(\hat{S}_H, T_i)$ .

*Proof.* If  $i = j$  then trivially  $op_j \in Vis(\hat{S}_H, T_i)$ . Otherwise:

$$i \neq j \wedge \text{Lemma 70} \implies \exists T_j \wedge op_j \in H_C|T_j \quad (584)$$

$$(584) \wedge \text{Lemma 85} \wedge res_i(C_i) \in H_C|T_i \implies \exists res_j(C_j) \in H_C|T_j \quad (585)$$

$$\text{Def. 37} \wedge (585) \implies res_j(C_j) \in \hat{S}_H|T_j \wedge T_j <_{\hat{S}_H} T_i \quad (586)$$

$$(586) \implies \hat{S}_H|T_j \subseteq Vis(\hat{S}_H, T_i) \implies op_j \in Vis(\hat{S}_H, T_i) \quad (587)$$

□

**Lemma 87.**  $\exists op_j = w_j(x)v \rightarrow ok_j \in LVis(\hat{S}_H, T_i)$ .

*Proof.* If  $i = j$  then trivially  $op_j \in LVis(\hat{S}_H, T_i)$ . Otherwise:

$$i \neq j \wedge \text{Lemma 70} \implies \exists T_j \wedge op_j \in H_C|T_j \quad (588)$$

$$(588) \wedge \text{Lemma 84} \wedge res_i(C_i) \in H_C|T_i \implies \text{either } \exists res_j(C_j) \in H_C|T_j \text{ or } \exists \ddot{op}_j^x \in H_C|T_j \quad (589)$$

$$\text{Def. 37} \wedge (589) \implies res_j(C_j) \in \hat{S}_H|T_j \wedge T_j <_{\hat{S}_H} T_i \quad (590)$$

$$(590) \implies \hat{S}_H|T_j \subseteq LVis(\hat{S}_H, T_i) \implies op_j \in LVis(\hat{S}_H, T_i) \quad (591)$$

$$(589) \implies \ddot{op}_j^x \in \hat{S}_H|T_j \wedge T_j <_{\hat{S}_H} T_i \quad (592)$$

$$(592) \wedge \hat{S}_H|T_j \subseteq LVis(\hat{S}_H, T_i) \implies op_j \in LVis(\hat{S}_H, T_i) \quad (593)$$

□

**Lemma 88.** *Given  $\hat{S}_H$  and any two transactions  $T_i, T_j \in \hat{S}_H$  s.t. there is an operation execution  $w_j(x)v \rightarrow ok_j \in \hat{S}_H|T_j$  and  $r_i(x) \rightarrow v \in \hat{S}_H|T_i$  then there is no operation  $w_k(x)u \rightarrow ok_k$  (executed by some  $T_k \in \hat{S}_H$ ) in  $Vis(\hat{S}_H, T_i)$  s.t.  $w_k(x)u \rightarrow ok_k$  precedes  $r_i(x) \rightarrow v$  in  $Vis(\hat{S}_H, T_i)$  and follows  $w_j(x)v \rightarrow ok_j$  in  $Vis(\hat{S}_H, T_i)$ .*

*Proof.* For the sake of contradiction, assume that  $op_k$  exists as specified.

If  $k = i$ , then  $op_k <_{H|T_i} op_i$ , which contradicts Lemma 70 (assuming unique writes).

If  $k = j$ , then from Lemma 84  $T_j$  is either committed or decided on  $x$  in  $\hat{S}_H$ . If  $T_i$  commits, then  $op_i$  reading  $v$  contradicts Lemma 70. If  $T_i$  does not commit in  $P$ , then this contradicts Lemma 85.

Otherwise,  $\exists T_k \in H$  s.t.  $op_k \in H|T_k$  from Lemma 84  $T_j$  is either committed or decided on  $x$  in  $\hat{S}_H$  and from Lemma 85  $T_k$  is committed in  $H$ . Since  $T_k$  commits, this contradicts Lemma 70. □

**Lemma 89.** *Given  $\hat{S}_H$  and any two transaction  $T_i, T_j \in \hat{S}_H$  s.t. there is an operation execution  $w_j(x)v \rightarrow ok_j \in \hat{S}_H|T_j$  and  $r_i(x) \rightarrow v \in \hat{S}_H|T_i$  then there is no operation  $w_k(x)u \rightarrow ok_k$  (executed by some  $T_k \in \hat{S}_H$ ) in  $LVis(\hat{S}_H, T_i)$  s.t.  $w_k(x)u \rightarrow ok_k$  precedes  $r_i(x) \rightarrow v$  in  $Vis(\hat{S}_H, T_i)$  and follows  $w_j(x)v \rightarrow ok_j$  in  $Vis(\hat{S}_H, T_i)$ .*

*Proof.* By analogy to Lemma 88. □

**Lemma 90.** *Any SVA history  $H$  is final-state last-use opaque.*

*Proof.* Given  $\hat{S}_H$ , let  $T_i \in \hat{S}_H$  be any transaction that is committed in  $\hat{S}_H$ . In that case, from Lemma 86 and Lemma 88, every read operation execution  $r_i(x) \rightarrow v$  in  $Vis(\hat{S}_H, T_i)$  is preceded in  $Vis(\hat{S}_H, T_i)$  by a write operation execution  $w_j(x)v \rightarrow ok_j$  (for some  $T_j$ ). In addition, from Assumption 1, every write operation execution  $w_i(x)v \rightarrow ok_i$  in  $Vis(\hat{S}_H, T_i)$  trivially writes  $v \in D$ . Therefore, for every variable  $x$ ,  $Vis(\hat{S}_H, T_i)|x \in Seq(x)$ , so  $Vis(\hat{S}_H, T_i)$  is legal. Consequently  $T_i$  in  $\hat{S}_H$  is legal in  $\hat{S}_H$ .

Given the same  $\hat{S}_H$ , let  $T_i \in \hat{S}_H$  be any transaction that is not committed in  $\hat{S}_H$  (so it is aborted in  $\hat{S}_H$ ). From Lemma 87 and Lemma 89, every read operation execution  $r_i(x) \rightarrow v$  in  $LVis(\hat{S}_H, T_i)$  is preceded in  $LVis(\hat{S}_H, T_i)$  by a write operation execution  $w_j(x)v \rightarrow ok_j$  (for some  $T_j$ ). In addition, from Assumption 1, every write operation execution  $w_i(x)v \rightarrow ok_i$  in  $LVis(\hat{S}_H, T_i)$  trivially writes  $v \in D$ . Therefore, for every variable  $x$ ,  $LVis(\hat{S}_H, T_i)|x \in Seq(x)$ , so  $LVis(\hat{S}_H, T_i)$  is legal. Thus,  $T_i$  in  $\hat{S}_H$  is last-use legal in  $\hat{S}_H$ .

Since all committed transactions in  $\hat{S}_H$  are legal in  $\hat{S}_H$  and since all aborted transactions in  $\hat{S}_H$  are last-use legal in  $\hat{S}_H$ , then, by Def. 21  $H$  is final-state last use opaque.  $\square$

**Theorem 22.** *Any SVA history  $H$  is last-use opaque.*

*Proof.* Since by Lemma 90 any SVA history  $H$  is final-state last-use opaque, and any prefix  $P$  of  $H$  is also an SVA history, then every prefix of  $H$  is also final-state last-use opaque. Thus, by Def. 22,  $H$  is last-use opaque.  $\square$